Bernhard Bauer

Attributed Algebraic Specifications

Herbert Utz Verlag Wissenschaft
München 1996

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Bauer, Bernhard:
Attributed algebraic specifications / Bernhard Bauer. -
München : Utz, Wiss., 1996
(Informatik)
Zugl.: München, Techn. Univ., Diss., 1996
ISBN 3-89675-147-6

ISBN 3-89675-147-6

Printed in Germany

Druck: Komplan R. Biechteler KG, München
Bindung: drucken + binden gmbh, München

Herbert Utz Verlag Wissenschaft, München
Tel.: 089/3077-8821 - Fax: 089/3077-9694

Abstract

In this thesis a new specification technique, called *attributed algebraic specification*, is investigated closing the gap between the specification formalisms of algebraic specifications and attribute grammars to combine the advantages of algebraic specifications (e.g. precise model class semantics, theorem proving techniques, deductive aspects, abstraction, refinement relations) and those of attribute grammars (e.g. intuitivity, efficiency, description of context dependent information, distinction of syntax and semantics specification). The contribution of this thesis is the extension of algebraic specifications in such a way that the ideas of both specification techniques are combined and extended, i.e. especially to describe context dependent informations and to prove their correctness.

Undirected attribute equations, instead of directed attribute equations as in usual attribute grammar systems, are allowed for an abstract specification of the attribute dependencies in the proposed specification technique. These equations are solvable in the considered logical framework. But they require the investigation of new dependency notions and new attribute evaluation strategies.

For the new formalism *specification building operations* are defined and it is shown how they can be normalized. Especially these operations take the notion of behaviour for attributed trees as an abstraction mechanism into consideration.

A *standard* and a *behavioural implementation relation* are developed with proof theoretical characterizations and properties, like transitivity or monotonicity relative to the specification building operations.

Calculi for solving existentially and universally quantified formulae are presented. For the universally quantified formulae an induction principle and a notion of complete set is defined usable especially for showing the correctness of implementation relations.

Case studies show the applicability of the new approach in the area of specifying the dynamics of user interfaces, compilers and document architecture systems.

The presented formalism can be used in a formal *software engineering process*. In this framework a software engineering process starts with a loose attributed algebraic specification which will be refined until a usual attribute grammar is reached from which an executable program can be generated. In this process all aspects of the thesis can be applied as the following considerations show:

Undirected attribute equations allow a *loose* and *abstract* specification. To execute such specifications (catchword: *rapid prototyping*), calculi are necessary for solving existentially quantified formulae. To speed up execution time attribute *evaluation algorithms* have to be proposed for undirected attribute equations. Specification building operations allow *modularization* in the software development process resulting in readable specifications. To obtain *correct* software after several refinement steps first of all *testing on a high level of abstraction* is necessary (usage of calculi for proving existentially and universally quantified formulae). Secondly, the *correctness of the refinement steps* must be shown (proof theoretical handling of the refinement relations). In the considered case studies these aspects are shown exemplarily.

Acknowledgement

First of all, I would like to express my gratitude to my supervisor Prof. Dr. Jürgen Eickel for giving me the opportunity to investigate this research at his chair at the Munich University of Technology, for his guidance and his animating discussions. Furthermore for introducing me into the specification of user interfaces.

I am equally indebted to Prof. Dr. Martin Wirsing for his help and for the deliberations which contribute to the elimination of several technical deficiencies. Moreover, to act as the second expert. He has also attracted my interest in algebraic specifications and compiler construction during his lessons and my diploma thesis at the University of Passau.

Especially I want to thank Dr. Rolf Hennicker for many fruitful discussions on the topic of algebraic specifications and observability issues during the last few years and for the refinement of the theoretical considerations. Thanks to Dr. Ulrich Fraus for reading drafts of this thesis and discussions on the topic of the TIP system.

Special thanks go to all my colleagues at the chair of Prof. Dr. Jürgen Eickel for a good working atmosphere, a lot of discussion leading to new ideas, for reading drafts and constructive comments, namely (in alphabetical ordering) Riitta Höllerer, Frank Lonczewski, Dr. Christoph Pleier, Dr. Arnd Poetzsch-Heffter, Siegfried and Werner Schreiber. Thanks to Hans Wittner, Franz Haßmann and Mariane Hargitay for their help.

Moreover thanks to all those people who have directly or indirectly influenced this work.

Finally, I want to thank Barbara for good times and encouragement. My special thanks go to my parents, who have made it possible for me to study informatics and for their support during the whole time.

Table of Contents

1 Introduction

Attribute grammars as introduced in [Knuth 68] are a well accepted formalism, for e.g. specifying compilers [Aho et al. 86], language-based environments [Reps, Teitelbaum 84], user interfaces [Krönert et al. 89; Schreiber 94a, 94b, 96; Bauer 95], document architecture [Eickel 90; SchreiberW 96] and (static) semantics of programming languages [Aho et al. 86] (for more details on attribute grammars see [Deransart et al. 88; Alblas, Melichar 90]). The theory of attribute grammars and their application are well studied. Many extensions of pure attribute grammars can be found in the literature, e.g. attribute coupled grammars [Ganzinger, Giegerich 84], higher-order attribute grammars [Vogt et al. 89], declarative extended attribute grammars [Knopp 90; Liebl et al. 90], object-oriented extensions [Hedin 89, 92, 94] and formal specifications of context dependent syntax of programming languages [Poetzsch-Heffter 91a, 91b, 94, 96].

Moreover, a lot of attribute grammar systems have been implemented (cf. e.g. [Reps, Teitelbaum 84; Grosch 89; Magnusson et al. 90; Kastens 91; Gray et al. 92; Poetzsch-Heffter 96]). The advantages of attribute grammars are efficiency (i.e. efficient algorithms exist for computing the attribute values), intuitivity (i.e. the notion of attribute grammars is easy to understand), detailed investigation (i.e. a lot of research has been done on this topic), wide use (e.g. attribute grammars are a well accepted technique for specifying compilers). Furthermore, attribute grammars are a highly declarative description which can be translated into efficient imperative programs.

On the other hand *algebraic specifications* (cf. e.g. [Ehrig, Mahr 85; Wirsing 90]) are used for the development of software systems (starting with [Liskov, Zilles 74; Guttag 75; ADJ 76]), since stepwise refinement and structured programming introduced by Dijkstra and Wirth are supported. These specifications are used frequently for the stepwise refinement of high-level specifications to low-level programs or executable specifications (rapid prototyping). Especially algebraic specifications are used to obtain correct software. They allow the description of data structures and functions in an implementation independent way. An algebraic axiomatic specification consisting of a signature and the characteristic properties defines a class of algebras. Concerning refinement or implementation steps abstraction mechanisms are realized as notions of behaviour [Goguen, Meseguer 82; Reichel 81; Sannella, Wirsing 83; Wirsing 86; Bidoit et al. 94].

The main contribution of this thesis is to extend algebraic specifications in such a way that the ideas of algebraic specifications and attribute grammars are combined.

Our efforts are visualized in figure 1.

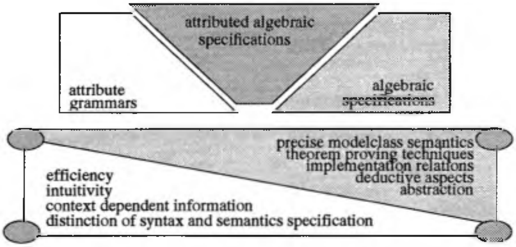

attributed algebraic specifications
attribute grammars
algebraic specifications
precise modelclass semantics
theorem proving techniques
implementation relations
deductive aspects
abstraction
efficiency
intuitivity
context dependent information
distinction of syntax and semantics specification

figure 1: combining attribute grammars and algebraic specifications

The proposed specification technique, called *attributed algebraic specification*, closes the gap between the specification formalisms of attribute grammars and algebraic specifications. The advantages of algebraic specifications, namely a precise model class semantics, theorem-proving techniques, deductive aspects, abstraction, refinement or implementation relations, and those of attribute grammars, namely intuitivity, efficiency, context dependent information, distinction of the syntax and semantics specification are obtained. Roughly speaking the shortcomings of the one approach are the advantages of the other approach and vice versa. Therefore the new technique subsumes the advantages of both techniques, neglecting more or less the shortcomings of both. In this framework a software engineering process starts with an attributed algebraic specification - being an extension of pure algebraic specifications. This specification will be refined until a specification is reached - being a usual attribute grammar - which can be used as a basis for generating an executable program. The following subsections give an overview of the investigated work.

1.1 Extensions of Algebraic Specifications

Pure algebraic specifications are extended to describe context dependent information and to specify its properties.

On the syntactical side the notion of term is extended to *attribute term* and formulae are defined over these attribute terms.

On the semantical side the interpretation function has to adapted to the new term notion.

Attribute terms allow the definition of context dependent information in the following way:

- A set of *occurrence terms* is associated with a usual ground constructor term, i.e. a term consisting only of constructor symbols and no identifiers, denoting the occurrences of the term, i.e. viewing a term as a tree, the nodes of the tree.
- Distinguished function symbols, namely *attribute function symbols*, can be applied to these occurrence terms representing the information to be assigned to the nodes of the tree. The application of an attribute function symbol to an occurrence term is called *attribute occurrence*.
- For technical reasons (e.g. for the proof principle, automatic term construction, implementation proofs, attribute evaluation,...) and to describe remote access of attribute occurrences a new kind of identifier is introduced, called *subterm identifier*, which can be substituted using any context in the sense of algebraic specifications.

Thus an attributed algebraic specification consists of a signature with a set of sort, constructor and function symbols with a distinguished subset of attribute function symbols and a set of axioms being equations over the extended term notion. In a logical framework existentially quantified formulae can be solved, therefore undirected attribute equations specifying only some relations on the attribute occurrences can be used.

For attributed algebraic specifications the same structuring mechanisms are defined as in ASL [Wirsing 86]. Moreover, these structured specifications can be normalized like ASL-operations.

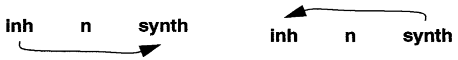
1.2 Notion of Behaviour

In the framework of algebraic specifications the notion of behaviour has been proven to be an adequate mechanism of abstraction. In the framework of attribute grammars no abstraction mechanism can be found. But considering e.g. the specification of a compiler abstraction is useful, too. Usually the compilation process is split into several phases with one attribute grammar for each phase. In each phase more or less one attribute is interesting, e.g. in the type analysis phase it is the attribute in which the type of an expression is stored and in the code generation phase it is the attribute containing the calculated code. But auxiliary attributes are necessary for deriving the type of an expression or the code (see the case study of the compiler). Therefore attribute grammars can be viewed as behaviourally equivalent, if the derived type or code is equivalent or behaviourally equivalent. Behavioural equivalence on attribute values is useful, since e.g. optimized and unoptimized code should be equivalent. Thus the following notion of behaviour for attributed algebraic specifications is obtained:

> Two attributed trees are *behaviourally equivalent*, if the values of the corresponding observable attribute occurrences are behaviourally equivalent in the sense of algebraic specifications.

1.3 Attribute Dependencies and Attribute Evaluation

In attribute grammar systems directed dependency graphs are used for analysing the attribute dependencies and as a basis of the generation of efficient attribute evaluators. Since attributed algebraic specifications allow *undirected* attribute equations (instead of directed attribute equations as in attribute grammars) specifying only some relations on the attribute occurrences *dependency sets* are defined for the description of the reciprocal dependence of attribute occurrences. These dependency sets are the basis for the determination of the attribute evaluation ordering. In contrast to usual attribute grammars it is *not* possible to determine the subordinate and superior characteristic set analogously to graphs and use this knowledge to derive the attribute evaluation ordering. These characteristic sets cannot be determined without the knowledge of the attribute evaluation ordering. Let us consider an example showing this problem:



Having a node *n* with an inherited attribute *inh* and a synthesized attribute *synth* and an attribute equation *inh*(*n*) = *synth*(*n*), i.e. the inherited attribute value at node *n* is equal to the synthesized attribute at this node, it depends on the attribute evaluation ordering whether there is a subordinate (the inherited one is calculated before the synthesized one) or a superior relation (the synthesized one is calculated before the inherited one) between the two attribute occurrences. Therefore the attribute evaluation ordering has to be determined before the subordinate and superior characteristic set can be calculated. These sets are consequently not involved in determining the attribute evaluation ordering, but can be used as a basis for getting efficient heuristics of the calculi. Having determined such an ordering for each node of a tree the visit sequences can be computed. Having calculated the visit sequences the generation of attribute evaluators can be performed as for usual attribute grammars with the exception that a narrowing engine is necessary for the calculation of the values of the attribute occurrences, because it is dealt with undirected attribute equations. To use the specifications prototypically a dynamic attribute evaluation algorithm is developed to determine without a generation process the attribute evaluation ordering and the attribute values of a given tree.

1.4 Calculi

Extending the notion of terms to attribute terms the usual unification algorithm has to be adapted. Especially the use of subterm identifiers (being place holders for terms with special insertion places, like contexts in an algebraic specification) induces major revises to the existing unification calculi. For two unifiable attribute terms there cannot be determined a most general unifier, but a complete set of minimal unifiers. A new unification algorithm will be given, computing such a complete set of minimal unifiers of a set of terms iff a set of terms is unifiable. The correctness and completeness of the unification calculus is shown. The obtained complete set of minimal unifiers is finite.

Since we are working with attributed algebraic specifications in a logical framework, it is possible to perform deductions. The values of the attribute occurrences specified using undirected attribute equations can be calculated solving existentially quantified formulae.

In particular, it is possible to solve formulae of the form

$$\exists\, sv_{s_1, s_2, \ldots, s_n \rightarrow s}.\ P(sv)$$

with a subterm identifier *sv* and a property *P* containing the subterm identifier *sv*, i.e. solutions of *sv* should be derived, such that *P* is valid. An application of deduction is in the framework of automatic programming: Let the static semantics of a programming language be specified using an attributed algebraic specification. Solving such formulae a program fragment can be completed such that a correct program is obtained relative to the semantics definition. E.g. variables can be declared automatically or type information of functions can be inserted automatically. Another application is the derivation of intelligent help in the framework of user interfaces: E.g. having a given state of the application and an already performed dialogue the necessary dialogue steps can be determined to reach another state of the application.

In this thesis a calculus is presented for solving existentially quantified formulae. Beyond existentially quantified formulae, universally quantified formulae are of interest. Therefore the equational calculus is extended to handle the new term notion and an induction principle is presented, called *attributed term induction*, to prove properties over attribute occurrences of occurrence terms of a distinguished sort. For this proof principle a semi-algorithm is developed, which was the basis of the implementation of the proof principle [Duschl 94; Weiß 95]. With the induction principle e.g. invariants can be shown which have to hold between attribute occurrences.

A generalization of attributed term induction is obtained introducing an induction ordering and a notion of *complete sets of occurrence terms* and *complete sets of subterm identifiers* (computable by the presented semi-algorithm). These notions allow to prove properties between occurrences of different sorts.

Efficient heuristics can be obtained taking the dependencies into consideration which are achieved in the analysing phase of the attributed algebraic specification.

1.5 Refinements

Two kinds of implementation relations for attributed algebraic specifications are investigated. One standard implementation relation which does not take behavioural aspects into consideration and a behavioural implementation notion based on the already presented idea of behaviour for attributed algebraic specifications. For both implementation notions we show the transitivity and monotonicity relative to the proposed specification building operations. In particular, proof theoretical characterizations of the standard and behavioural implementation relation are developed.

The composability of the implementation relations allows to implement the algebraic part, i.e. the part of an attributed algebraic specification, being a usual algebraic specification, independent of the attribution part, i.e. the part of the attributed algebraic specification defining attributions on constructor terms.

This property simplifies the verification effort for proving implementations. For example let Sp_1, Sp_2 and Sp_3 be usual algebraic specifications stored in an algebraic specification library. Suppose it was already shown that Sp_3 is an implementation of Sp_2 and Sp_2 is an implementation of Sp_1. In order to show that the enrichment of an algebraic specification Sp_1 by an attribution *Attr*, denoted by **enrich** Sp_1 **by** *Attr*, is implemented by **enrich** Sp_2 **by** *Attr* which in turn is implemented by **enrich** Sp_3 **by** *Attr* it is sufficient to know that

Sp_1 is implemented by Sp_2 and Sp_2 is implemented by Sp_3

Using transitivity and monotonicity it holds: **enrich** Sp_1 **by** *Attr* is implemented by **enrich** Sp_3 **by** *Attr*. Thus algebraic specification libraries can be used in the new approach without doing verification twice.

1.6 Related Work

The new specification technique is a combination of attribute grammars and algebraic specifications thus we have to discuss related work of these two topics.

In the framework of attribute grammars *higher-order attribute grammars* [Vogt et al. 89; Swierstra, Vogt 91] and *attribute coupled grammars* [Ganzinger, Giegerich 84] can be found. Roughly speaking syntax trees are first class citizens, i.e. syntax trees can be the result of an attribution and can be pasted into an incomplete syntax tree. Since in the new approach syntax trees are usual terms, which can be the result of a calculation, these extensions can also be expressed in attributed algebraic specifications.

Tree transformations [Alblas 89] for attribute grammars describe by rules the transformation of an attributed tree in a new attributed tree dependent on the values of attribute occurrences of the syntax tree. In the new approach equations can be defined on constructor terms defining such transformation rules.

Primitive recursive schemes [Courcelle, Franchi-Zannettacci 82] being a restricted class of algebraic specifications have been introduced to express attribute grammars in the framework of algebraic specifications. Techniques from attribute grammars are translated into algebraic specifications [Klint 93; Meulen 94; Deursen 94]. But this specification formalism needs a synthesized form of the attribute equations with the result that usual techniques from algebraic specifications cannot be applied, like implementation relations or proof principles. Thus with this technique only the advantages of attribute grammars are obtained for a restricted class of algebraic specifications and not vice versa.

Proof principles for attribute grammars are rarely found in the literature. The main contributions are [Katayama, Hoshino 81] and [Courcelle, Deransart 88]. The idea is to assign invariants to each non terminal of the syntax trees and to prove the correctness of the attribute grammar relative to the invariants and the attribute dependencies. Since the proof principle is based on the attribute dependency graph, it cannot be applied for attribute grammars with undirected attribute equations which are used in the new approach. Moreover, it is necessary to find invariants for each non terminal associated with nodes of a syntax tree making the proof much more complicated, since finding invariants is a hard work to do. These proof principles suffer from efficient heuristics because the proof has to be performed for all invariants.

In the proof principle of *context induction* [Hennicker 91], being an inspiration for the new proof principle, another ordering on contexts is used than in our approach. Furthermore, in the new approach it is combined with term induction.

Object-oriented extensions of attribute grammars are given in [Hedin 89, 92, 94]. The abstract syntax tree is modelled using objects in such a way that each production corresponds to a class definition. Using inheritance (in the object-oriented sense) a kind of order-sortedness is defined. The main aims of G. Hedin can be obtained using order-sorted signatures instead of usual signatures. Attributed algebraic specifications can be extended in such a way. In order to keep the formalism lean, especially for the definition of the calculi, order-sortedness was renounced in this thesis.

The *MAX system* (cf. e.g. [Poetzsch-Heffter 96]) and its formalism can be viewed as a first step embedding attribute grammars in a functional and algebraic framework. This formalism defines concrete algebras for specifying the occurrences of a tree and the trees themselves. Therefore new occurrence sort symbols are introduced beyond the usual sort symbols. It is referred to the occurrences of a term using selector functions on the arguments of a function. Attributes are viewed as functions and are specified in a functional way with an extended pattern-matching mechanism, in comparison to functional programming languages allowing to define context dependent informations. But there are no consideration for proving their correctness. Implementations or abstraction mechanisms are not supported. The output of the new approach can be a MAX specification which can be used as an input to generate an efficient program.

Comparing the new specification technique with *higher-order algebraic specifications* (cf. e.g. [Möller 87; Heering et al. 94; Kosiuczenko, Meinke 96]) the use of subterm identifiers is a very restricted application of higher-order algebraic specifications. But higher-order algebraic specifications do not concern context dependent informations.

Viewing attribute grammars as algebras [Chirica, Martin 79] explicit algebras are used instead of a class of algebras satisfying some properties. Moreover, the semantics is defined as the solution of the equational system obtained for a given derivation tree. They present a possibility converting an attribute grammar into its synthesized form.

Modularity and *reusability* issues for attribute grammars are considered e.g. in [Kastens, Waite 92]. In the framework of attribute grammars the notion of modularity is strongly connected with the possibility to define attribute dependencies independent of the underlying grammar. But e.g. in the area of algebraic specifications „real“ modularisation is meant, i.e. structuring mechanisms are considered for building specifications from simpler ones. The same aim is persued in the new technique and using subterm identifiers an abstraction from the underlying signature is obtained, too.

In usual attribute grammar systems it is dealt with directed attribute equations. For these directed attribute equations there exist elaborated attribute evaluation techniques, especially for handling *cyclic attribute dependencies* [Farrow 86; Jones 90; Walz, Johnson 95] and *incremental attribute evaluation* (cf. e.g. [Reps et al. 83]). Both aspects can be considered in the new technique, performing a fixpoint computation for the attribute values of cyclic dependencies and using the attribute dependencies and the visit sequences for incremental attribute evaluation.

1.7 Case Studies

Among other case studies considered for this thesis, three case studies are presented showing typical applications of the new specification technique.

One application area of the new approach can be seen in the specification and verification of the dynamics of user interfaces. This case study comes up from a research project with Siemens. Here the dynamics of the user interface for an ISDN telephone is given and some properties of the specification are shown. One property is shown using the proof principle of attributed term induction and other properties using the analysis techniques for attributed algebraic specifications, namely attributed signature flow analysis. The case study is more or less taken from [Bauer 95] and can be shown using the system implemented in [Duschl 94; Weiß 95]. Attributed algebraic specifications have been proven to be an adequate specification technique for the dynamics of user interfaces and showing their correctness. Applying the attributed narrowing calculus it is also shown how the derivation of intelligent help can be performed. In this case study moreover we study how the dynamics of user interfaces can be generated from an algebraic specification of the application [Bauer 96].

The next case study is taken from the framework of compiler construction. Here the compilation of expressions into stack machine code and (un)optimized register code is considered. It is shown that the compilation is semantics preserving. Afterwards the implementation relation of the compilation into unoptimized register code and into optimized register code is presented. Note, that the first translations, namely from expressions to stack machine code and unoptimized register code is not an implementation because a signature change is performed, whereas the other compilations describe a behavioural implementation from unoptimized to optimized register code.

Another typical application area of the new specification technique is its use in the area of document architecture. Here the problem of calculating the length of inner boxes is considered such that a given length of the whole box is reached. Implementations and attribute evaluation aspects for these specifications are taken into consideration.

These case studies explain by typical examples how real problems can be solved and that all aspects of the specification framework presented in this thesis are usable for them. The main aspects of this work and their application in the case studies (denoted by: X) are shown in table 1:

considered aspects	user interface specification	compiler specification	document architecture
remote access of attribute values	X	X	X
undirected equations			X
several correct attributions			X
observability issues	X	X	X
universally quantified formulae	X	X	X
existentially quantified formulae	X	X	
standard implementation			X
behavioural implementation		X	
structuring mechanisms	X	X	X
attributed signature flow analysis	X		

table 1: case studies show considered aspects

A detailed view is given in chapter 8.

1.8 Further Research

Up to now only some restricted implementations of the presented specification technique exist. The most elaborated implementation is a theorem prover for the presented proof principle of attributed term induction [Duschl 94; Weiß 95]. This theorem prover has shown some restrictions which have been overcome using complete sets and induction orderings. The system allows to show e.g. the properties of the ISDN telephone case study. Among the implementation of the attributed term induction the signature flow analysis problem of reachability is implemented.

It is planned to perform a prototypical implementation of the other aspects, especially implementation of the algorithms for narrowing and of the attribute evaluation algorithm in the functional programming language *Gofer*. First steps are already taken.

In usual attribute grammar systems the input is a file containing the text to be analysed. The scanner and parser transform the text file into an abstract syntax tree. This abstract syntax tree is attributed either while building the tree or after building the syntax tree. In the framework of user interface specification this proceeding is not sufficient, because the input is not a text file but a stream of tokens. Here parts of the syntax tree have to be built without knowing the complete input. In this case parts of the syntax tree or better called dialogue tree has to be transformed. Rewrite-rules can be defined which are applicable, if a special token is delivered from the user interface. These rewrite rules are comparable with state-based rewrite rules found in cf. Maude [Meseguer 93a, 93b].

Attribute grammars define attribute dependencies within a syntax tree. With subterm identifiers attribute dependencies between different constructor terms can be expressed. Since traces of processes can be defined using grammars (cf. e.g. [Hirshfeld et al. 96]), attribute grammars allowing attribute dependencies between different trees can be used to specify communication between different processes.

In the proposed approach it is dealt with equations, which can be extended to conditional equations.

Assuming a system for attributed algebraic specifications some more complex case studies can be considered, like a complete compiler for a small imperative or functional programming language. These examples would result in hints where the formalism has to be adapted.

With cyclic attribute dependencies the dynamics semantics of programming languages can be specified. But this implies the adaption of the attribute evaluation and attribute dependency strategies analogous to [Farrow 86; Jones 90]. In particular, the proof principles have to be extended to handle cyclic dependencies by introducing some fixpoint induction rule.

Concerning attribute evaluation aspects, incremental attribute evaluation can be performed to shorten execution time in a system for prototypical use of the specifications. Here the well known ideas from incremental attribute evaluation (cf. e.g. [Reps et al. 83]) can be used in the new specification technique. To speed up unification and attribute evaluation local attribute dependencies can be generated from the global attribute dependencies resulting in a more implementational look on the underlying attribute grammar.

Another point for further research is to add „specification sugar" as it can be found in the other attribute grammar systems, like e.g. inheritance in the object-oriented sense.

1.9 Résumé

This thesis shows how algebraic specifications can be extended in a uniform way to combine the advantages of algebraic specifications and attribute grammars, neglecting the shortcomings of both. In the new approach context dependent information can be defined in the same intuitive way as in attribute grammars.

The proposed specification technique can be used in a formal software engineering process starting with an abstract specification and arriving after several refinement steps at a usual attribute grammar. The correctness of the software can be guaranteed if the correctness of each refinement step is shown. For this purpose notions of implementation relations are introduced and proof theoretical characterizations are given. Especially, the new notion of behaviour based on an intuitive idea has been proven to be a good abstraction mechanism for attributed algebraic specifications. Starting with specifications where several design decisions are left open, undirected attribute equations are an essential component of the new technique.

The presented structuring mechanisms increase the re-use of specifications and allow to handle complex software projects. E.g. in the framework of compiler construction the problems of identification, typing and code optimization appear in nearly every compiler. The specifications for these problems can be proven correct and put into libraries. Since the implementation relations are monotone relative to the specification building operations such a program development is supported.

Several well known extensions of attribute grammars are contained in the new approach. In particular, attributed algebraic specifications have the advantages of other techniques neglecting sometimes the disadvantages of them.

The presented calculi can be used for verification purposes and for prototypical use of the specifications. Using the specifications in a prototypical way, efficient attribute evaluation strategies were developed. Since visit sequences can be calculated efficient programs can be generated from the specifications.

Each case study shows some interesting aspects of the considered new specification technique. Especially the non trivial compiler example of register placing and its optimization shows the usability of the calculi, proof principles and implementation relations.

This thesis opens new application areas for attribute grammars and algebraic specifications.

1.10 Organization of the Thesis

The rest of this thesis is organized as follows:

In chapter 2 the basic notions of algebraic specifications (section 2.1) and attribute grammars (2.2) are summarized.

Chapter 3 starts with a motivation and an introduction into the new specification formalism (section 3.1) and defines the syntax (section 3.2) and semantics (section 3.3) of the new technique afterwards. After investigating a notion of behaviour for attributed algebraic specifications in section 3.4, structuring mechanisms based on ASL are developed for attributed algebraic specifications in section 3.5. A technique for analysing the specifications is presented in section 3.6.

Attribute dependencies and attribute evaluation are the subject of chapter 4. Starting with an introduction into attribute dependencies and attribute evaluation for usual attribute grammars in section 4.1. Afterwards attribute dependencies and the attribute evaluation for specifications with undirected attribute equations are taken into consideration (section 4.2).

Calculi for attributed algebraic specifications are investigated in chapter 5. A unification algorithm for the extended term notion is given in section 5.1. To prove universally quantified formulae a calculus is given in section 5.2 and to solve existentially quantified formulae a calculus is presented in section 5.3.

To use the new specification technique in a formal software engineering process notions of implementation relations have to be developed (chapter 6). The aims are stated in section 6.1. Section 6.2 and section 6.3 investigate a notion of standard and behavioural implementation relation, respectively. An example of an implementation proof is given in section 6.4. Properties of the implementation relations are discussed in section 6.5.

Chapter 7 compares the new approach with the related work in the literature. Since the new technique is a combination of attribute grammars and algebraic specifications, the differences are discussed in section 7.1 (attribute grammars) and section 7.2 (algebraic specifications).

Case studies showing all aspects of the specification technique are given in chapter 8, namely how to specify the dynamics of user interfaces (section 8.1), compilers (section 8.2) and document architecture systems (section 8.3).

Chapter 9 deals with further research directions. A prototypical implementation of the presented theory is outlined in section 9.1. The use of state-based rewriting is shown in section 9.2. In section 9.3 it is discussed how communication between different processes can be handled and it is shown how communication can be defined using conditional equations. Section 9.4 deals with the topic of user interface. Ideas for further case studies are given in section 9.5 and hints for extensions concerning the attribute evaluation and the presented calculi are described in section 9.6. The chapter ends with some comments on „specification sugar“ to write specifications in a more elegant way (section 9.7).

Concluding remarks are given in chapter 10.

2 Basic Notions

In this section we briefly summarize the basic notions of algebraic specifications and attribute grammars which will be used in the following (for more details see e.g. [Ehrig, Mahr 85; Wirsing 90; Deransart et al. 88; Wilhelm, Maurer 92]).

2.1 Algebraic Specifications

For more details on algebraic specifications see e.g. [Ehrig, Mahr 85; Wirsing 90]. The given definitions are based on [Wirsing 90].

A *signature* $\Sigma = (S, C, F)$ consists of a non empty set S of sort symbols and non empty $(S^* \times S)$-indexed sets C and F of constructor and function symbols, respectively.

A functionality $fct(f) = f\colon s_1, s_2,\ldots, s_n \rightarrow s$ is associated with every symbol $f \in C \cup F$ where $s_1, s_2,\ldots, s_n, s \in (S^* \times S)$. The following selector functions are defined to extract the sort, functions, constructor and the operation symbols, i.e. function and constructor symbols, of a signature:

$sorts(\Sigma) = S$, $cons(\Sigma) = C$, $funcs(\Sigma) = F$, $opns(\Sigma) = C \cup F$, $cons_s(\Sigma)$ are the constructors of Σ of sort s.

Let $\Sigma = (S, C, F)$ be a signature and $X = (X_s)_{s \in S}$ a family of sets X_s of identifiers of sort $s \in S$. X, S and F are pairwise distinct. The set of Σ-*terms* (for short: *terms*) of sort s with identifiers in X is denoted by $T_\Sigma(X)_s$ and is inductively defined by:

(1) each identifier $x \in X_s$ is a Σ-term of sort s.

(2) if $t_1, t_2,\ldots, t_n$ are Σ-terms of sort $s_1, s_2,\ldots, s_n$ $(n \geq 0)$ and $(f\colon s_1, s_2,\ldots, s_n \rightarrow s) \in C \cup F$, then $f(t_1, t_2,\ldots, t_n)$ is a Σ-term of sort s.

The set of Σ-terms with identifiers in X is denoted by $T_\Sigma(X)_{s \in S}$ and abbreviated by $T_\Sigma(X)$.

If $X = \varnothing$ then $T_\Sigma(\varnothing)$ is abbreviated by T_Σ and $t \in T_\Sigma$ is called *ground term*.
$var(t)$ denotes the set of identifiers in t.

Let $\Sigma = (S, C, F)$ be a signature and $X = (X_s)_{s \in S}$ be an S-indexed set of sets of identifiers. The set of (well formed) Σ-*formulae* is inductively defined by:

(1) If $t, r \in T_\Sigma(X)_s$ are attribute terms of sort s, then $t = r$ is a Σ-formula (called *equation*),

(2) If Φ, Ψ are Σ-formulae, then $\neg\Phi$ and $\Phi \wedge \Psi$ are Σ-formulae,

(3) If Φ is a Σ-formula, then $\forall\, x_s.\ \Phi$ is a Σ-formula.

All other logical operations such as disjunction $\vee$, implication $\Rightarrow$ and the existential quantifier $\exists$ are defined as usual.

A Σ-*context* is any term $c[z_s]$ over the signature Σ containing a distinguished identifier z_s of some sort $s \in S$ such that z_s occurs exactly once in $c[z_s]$. The application of a context $c[z_s]$ to a term $t \in (T_\Sigma)_s$ is defined by substituting the context identifier z_s by t. To shorten notation $c[\, z_s / t\,]$ is abbreviated by $c[t]$. In particular, for any sort s, the identifier z_s is a

Σ-context (called *trivial context*) of sort s and it holds $z_s[t] = t$. A behavioural context is a context of behavioural sort $S_{Obs} \subseteq S$.

A *(partial)* Σ*-algebra* $A = ((A_s)_{s \in S}, (f^A)_{f \in C \cup F})$ consists of a family of carrier sets $(A_s)_{s \in S}$ and a family of (partial) functions $(f^A)_{f \in C \cup F}$ such that $f^A: A_{s_1}, A_{s_2},\ldots, A_{s_n} \to A_s$ if f has functionality $s_1, s_2,\ldots, s_n \to s$ (if the arity of f is zero then f^A denotes a constant object of A_s).

In this presentation we assume that $A_s \neq \varnothing$ for all $s \in S$ (for a discussion of empty carrier sets see [Goguen, Meseguer 82; Padawitz, Wirsing 84]).

If $X = (X_s)_{s \in S}$ is an S-indexed set of sets of identifiers and $A = ((A_s)_{s \in S}, (f^A)_{f \in C \cup F})$ a Σ-algebra, then a family of mappings $v = (v_s: X_s \to A_s)_{s \in S}$ is called *valuation* of X.

Let $v = (v_s: X_s \to A_s)_{s \in S}$ be a valuation for identifiers and A a Σ-algebra of the form $((A_s)_{s \in S}, (f^A)_{f \in C \cup F})$ with $\Sigma = (S, C, F)$. Then the *interpretation* for terms of $T_\Sigma(X)_{s \in S}$ wrt. v is a family of mappings

$$I^A_v = (I^A_{v,s}: T_\Sigma(X)_s \to A_s)_{s \in S}$$

defined by

(1) for each $x \in X_s$ holds: $I^A_{v,s}[x] = v_s(x)$.

(2) $I^A_{v,s}[f(t_1, t_2,\ldots, t_n)] = f^A(I^A_{v,s_1}[t_1], I^A_{v,s_2}[t_2],\ldots, I^A_{v,s_n}[t_n])$ for each $t_i \in T_\Sigma(X)_{s_i}$ $(1 \le i \le n)$ and $(f: s_1, s_2,\ldots, s_n \to s) \in C \cup F$, $f^A: A_{s_1}, A_{s_2},\ldots, A_{s_n} \to A_s$.

Let $\Sigma = (S, C, F)$ be a signature and A, B be Σ-algebras.

A family of mappings $h = (h_s: A_s \to B_s)_{s \in S}$ is called Σ*-homomorphism*, iff for all $(f: s_1, s_2,\ldots, s_n \to s) \in C \cup F$ and for all $a_1 \in A_{s_1}, a_2 \in A_{s_2},\ldots, a_n \in A_{s_n}$ holds:

$$h_s(f^A(a_1, a_2,\ldots, a_n)) = f^B(h_{s_1}(a_1), h_{s_2}(a_2),\ldots, h_{s_n}(a_n))$$

The interpretation of terms is for a given valuation v and an algebra A the unique Σ-homomorphic extension of v to $T_\Sigma(X)$.

For any Σ-algebra A over a signature (S, C, F), valuation $v = (v_s: X_s \to A_s)_{s \in S}$ and Σ-formula Φ the relation A *satisfies* Φ *wrt.* v (written $A, v \models \Phi$) is defined by:

(1) $A, v \models t = r$ holds, if $I^A_v[t] = I^A_v[r]$,

(2) $A, v \models \neg \Phi$ holds, if $A, v \models \Phi$ does not hold,

(3) $A, v \models \Phi \wedge \Psi$ holds, if $A, v \models \Phi$ and $A, v \models \Psi$ holds,

(4) $A, v \models \forall x_s.\ \Phi$ holds, if for all valuations v' with $v(x') = v'(x')$ for all $x_s \neq x'$, $A, v' \models \Phi$ holds,

with $t, r \in T_{(S, C, F)}(X)$ and Σ-formulae Φ, Ψ.

A satisfies Φ (written: $A \models \Phi$) iff for all valuations v holds: $A, v \models \Phi$.

A satisfies behaviourally $t = r$ (written: $A \models_{\text{beh}} t = r$) iff for all behavioural contexts $c[z_s]$

holds: $A \models \forall\, var(c).\ c[t] = c[r]$.

Let $\Sigma = (S, C, F)$ be a signature and Ax a set of Σ-equations of the form $t = r$ with $t, r \in T_\Sigma(X)_s$ for some $s \in S$. An *algebraic specification* is a pair $<\Sigma, Ax>$. The semantics is described by its signature

$$sig(<\Sigma, Ax>) = \Sigma$$

and its class of models

$$Mod(<\Sigma, Ax>) = \{\, A \in Alg(\Sigma) \,\big|\, A \models ax \text{ for all } ax \in Ax \,\}.$$

A *signature morphism* $\sigma\colon \Sigma \to \Sigma_1$ between a signature $\Sigma = (S, C, F)$ and a signature $\Sigma_1 = (S_1, C_1, F_1)$ consists of mappings $\sigma_{sort}\colon S \to S_1$ and $\sigma_{func}\colon C \cup F \to C_1 \cup F_1$, such that for all $f \in C \cup F$ with $fct(f) = f\colon s_1, s_2, \ldots, s_n \to s$ holds:
$\sigma_{func}(f)$ has the functionality $fct(\sigma_{func}(f)) = \sigma_{sort}(s_1), \sigma_{sort}(s_2), \ldots, \sigma_{sort}(s_n) \to \sigma_{sort}(s)$, i.e. the renaming of the operation symbols is consistent with the renaming of the sort symbols.

A *substitution* $\sigma\colon X \to T_\Sigma(X)$ is a family of mappings $(\sigma_s\colon X_s \to T_\Sigma(X)_s)_{s \in S}$. For any term $t \in T_\Sigma(X)$, the *instantiation* $\sigma(t)$ is defined by simultaneously replacing all identifiers $x_1, x_2, \ldots, x_n \in X$ occurring in t by the terms $\sigma(x_1), \sigma(x_2), \ldots, \sigma(x_n)$. Substitutions are denoted by $[\, x_1 / \sigma(x_1), x_2 / \sigma(x_2), \ldots, x_n / \sigma(x_n) \,]$ if the domain of σ is $\{\, x_1, x_2, \ldots, x_n \,\}$.

A substitution $\sigma\colon X \to T_\Sigma$ is called *ground substitution*.

Let $\sigma\colon \Sigma \to \Sigma_1$ be a signature morphism.
The σ-*reduct* of an Σ-algebra $A = ((A_s)_{s \in sorts(\Sigma)}, (f^A)_{f \in opns(\Sigma)})$, written $A|_\sigma$, is the Σ_1-algebra with the carrier sets $(A|_\sigma)_s = A_{\sigma(s)}$ and functions $f^{A|\sigma} = \sigma(f)^A$.

Let $\Sigma = (S, C, F)$ and $\Sigma_1 = (S_1, C_1, F_1)$ be signatures with $\Sigma_1 \subseteq \Sigma$ and Σ-algebra $A = ((A_s)_{s \in S}, (f^A)_{f \in C \cup F})$, then $A|_{\Sigma_1} = ((A_s)_{s \in S_1}, (f^A)_{f \in C_1 \cup F_1})$.

2.2 Attribute Grammars

In this section we briefly summarize the basic notions of attribute grammars (for more details see e.g. [Deransart et al. 88]). We follow here more or less [Wilhelm, Maurer 92].

With context free grammars the syntactical structure of e.g. a programming language can be defined:

A *context free grammar* is a tuple $G = (N, T, P, S)$, with N, T finite alphabets, whereby N is the set of *non terminals*, T is the set of *terminals*, $P \subseteq N \times (N \cup T)^*$ is the set of production rules and $S \in N$ is the *axiom* or *start symbol*. The p-th production is denoted by $p\colon X \to w$ with $X \in N$ and $w \in (N \cup T)^*$.

An *attribute grammar AG* over a context free grammar G consists of the following components:

(1) Two distinct sets are associated with each symbol $X \in N \cup T$, namely the set of *inherited* attributes, denoted by $Inh(X)$, and the set of *synthesized* attributes, denoted by $Syn(X)$. The set of attributes of a symbol X, denoted by $Attr(X)$, is defined as the union of the inherited and synthesized attributes, i.e. $Attr(X) = Inh(X) \cup Syn(X)$. If

$a \in Attr(X_i)$ $(0 \leq i \leq n_p)$ then a has an *occurrence* in production $p{:}\ X_0 \rightarrow X_1\, X_2 \ldots X_{n_p}$ at the occurrence of X_i, denoted by a_i. $Occ(p)$ is the set of attribute occurrences in production p.

$Inh = \cup_{X \in N \cup T} Inh(X)$, $Syn = \cup_{X \in N \cup T} Syn(X)$, $Attr = Inh \cup Syn$

(2) For each $a \in Attr$ we associate a *domain* D_a which is the sort of the attribute value.

(3) For each $a \in Inh(X_i)$ with $1 \leq i \leq n_p$ and for each $a \in Syn(X_0)$ of a production rule $p{:}\ X_0 \rightarrow X_1\, X_2 \ldots X_{n_p}$ we define a *semantic rule*:

$a_i = f_{p,\,a,\,i}(b^1_{j1}, b^2_{j2}, \ldots, b^k_{jk})$ with $0 \leq j_l \leq n_p$, $1 \leq l \leq k$

such that $f_{p,\,a,\,i}$ is a function with functionality $D_{b1}, D_{b2}, \ldots, D_{bk} \rightarrow D_a$.

Let AG be an attribute grammar and $p{:}\ X_0 \rightarrow X_1\, X_2 \ldots X_{n_p}$ be a production. The attribute occurrence a_i with $a \in Inh(X_i)$ and $1 \leq i \leq n_p$ and with $a \in Syn(X_0)$ are called *defining occurrence* of an attribute. Otherwise it is called *applied occurrence. AG* is called in *normal form,* if all arguments of a semantic rule are applied occurrences.

The production local dependency relation $Dp(p) \subseteq Occ(p) \times Occ(p)$, denoted by $\rightarrow_{\text{local}}$, of a production p is defined by:

$b_j \rightarrow_{\text{local}} a_i$, iff $a_i = f_{p,\,a,\,i}(\ldots, b_j, \ldots)$ for a semantic rule of p.

An occurrence of the attribute b at X_j is in relation with an occurrence of a at X_i or a_i depends on b_j, if b_j is an argument in the semantical rule of a_i. The visualization of the relation is performed using a graph, called *production local dependency graph.*

Let t be a tree of the underlying context free grammar. The *individual dependency graph* on the attribute occurrences of t, denoted by $DGraph(t)$, is obtained putting together the production local dependency graphs of the productions used in t.

An attribute grammar is called *cycle free,* if $DGraph(t)$ contains for all trees of the underlying context free grammar no cycles.

An attribute grammar is *well formed* if it is cycle free.

Let t be a tree of the underlying context free grammar such that the root is marked with X. Restricting the transitive closure of $DGraph(t)$ to the attribute occurrences of the root, a relation $DGraph_{sub}(t) \subseteq Inh(X) \times Syn(X)$ is obtained, whose graph we call the *subordinate characteristic graph* of X induced by t. There is an edge from $a \in Inh(X)$ to $b \in Syn(X)$ in $DGraph_{sub}(t)$ if there is a way from a to b in $DGraph(t)$ in the considered subordinate tree fragment.

Let n be an inner node of a tree t marked with the non terminal $X \in N$. Let us consider the superior tree fragment of t at n, by $t \setminus n$ the tree is denoted obtained from t by deleting the subtree at n without n itself. The restriction of the transitive closure of $DGraph(t \setminus n)$ to the attribute occurrences of n defines a relation $DGraph_{sup}(X) \subseteq Syn(X) \times Inh(X)$, whose graph we call the *superior characteristic graph* at n for X induced by t. There is an edge from $a \in Syn(X)$ to $b \in Inh(X)$ in $DGraph_{sup}(t)$ if there is a way from a to b in $DGraph(t)$ in the considered superior tree fragment.

3 Attributed Algebraic Specifications

In this chapter a *motivation* for the new specification technique of attributed algebraic specifications is given. Afterwards the *syntax* and *semantics* of pure attributed algebraic specifications are defined. After developing a notion of *behaviour* for the new approach, *structured attributed algebraic specifications* are introduced to define complex specifications from smaller ones and a possibility is described to *normalize* them. The notion of *attributed signature flow analysis* is defined to analyse attributed algebraic specifications.

3.1 Motivation

The starting point for the new approach is the idea to embed attribute grammars into the notion of algebraic specifications to obtain the advantages of both specification techniques, because the advantages of the one formalism are more or less the shortcomings of the other one and vice versa shown in figure 2 (repeated from the introduction).

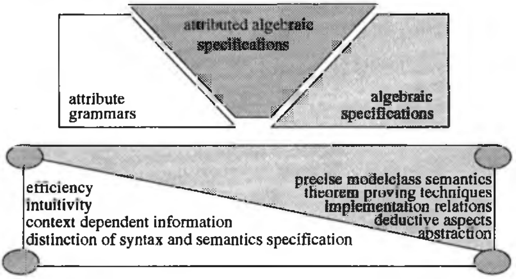

attributed algebraic specifications
attribute grammars
algebraic specifications
efficiency
intuitivity
context dependent information
distinction of syntax and semantics specification
precise modelclass semantics
theorem proving techniques
implementation relations
deductive aspects
abstraction

figure 2: combining attribute grammars and algebraic specifications

A combination of both approaches allows

- the specification of context dependent information,
- the distinction between the syntax and semantics specification,
- the efficient implementation using usual attribute grammars,
- the use of abstraction mechanisms like behaviour,
- the use of *undirected* attribute equations and
- a precise modelclass semantics with performing deductions and proving implementation relations,

because of the notion of attribute grammars (first three items) and algebraic specifications (rest of the items).

The components of an attribute grammar are:

- a context free grammar for describing the syntax,
- attributes assigning special informations to each node of a syntax tree,
- *directed* attribute equations defining functional dependencies between attribute occurrences and
- *external* semantical sorts and functions used in the attribute equations.

Pure algebraic specifications have the following components:

- a signature with sort, constructor and function symbols and

- a set of axioms specifying the semantics of the constructor and function symbols.

Since

- context free grammars can be expressed by signatures (cf. e.g. [Chirica, Martin 76, 79][1]),
- attributes can be viewed as functions,
- directed attribute equations are special equations and
- external sorts and functions can be seen as a usual algebraic specification,

it suggests itself to extend algebraic specifications in such a way that context dependent information can be expressed.

Let us consider a mobile as a running example (visualized in figure 3).

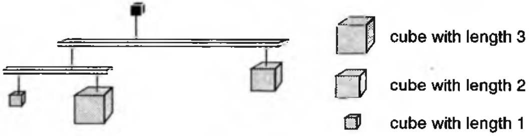


figure 3: mobile

This mobile can be seen in an abstract way as a tree/term shown in figure 4.

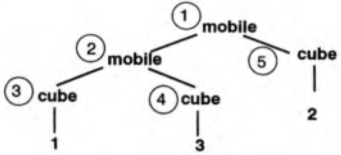


figure 4: mobile as a tree/term

A term representation for this mobile could be:

mobile(*mobile*(*cube*(1), *cube*(3)), *cube*(2))

Furthermore, special informations (attributes) can be assigned to the nodes of the mobile. Possible attributes are *weight*, describing the weight of the cubes in the submobiles below the actual node, *leftlength* and *rightlength*, describing the left length and right length from the fixing of a submobile, *length*, describing the length of a floor of a submobile, and *depth*, the depth of a submobile. The values of the attribute occurrences of the weight attribute can be specified using recursive equations[2]:

weight(*cube*(*l*)) = *l*,

weight(*mobile*(*m*1, *m*2)) = *weight*(*m*1) + *weight*(*m*2).

with identifier *l* denoting an arbitrary length *l* of a cube and identifiers *m*1 and *m*2 for arbitrary mobiles.

But the equations for the context dependent information *depth* cannot be described in

[1] Therefore the notion *term* and *tree* are used adequately in this thesis.

[2] The following simplifications concerning the weight of a mobile are made:
The weight of a cube is the length of the cube. Only the weight of the cubes is considered, i.e. the weight of the other mobile elements is neglected.

such a way. Here another strategy has to be investigated.

Informations are assigned to nodes of a tree. These *occurrences* have to be distinguished in the corresponding term. Therefore a set of *occurrence terms* is associated with a given constructor term denoting its occurrences. For each sort a distinguished constructor symbol occ_s: $s \rightarrow s$ is introduced to denote such occurrences. If the sort s is obvious usually *occ* is written instead of occ_s.

Given the constructor term *mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)) the associated occurrence terms are

occ(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2))), denoting node ① of the tree in figure 4,

mobile(**occ**(*mobile*(*cube*(1), *cube*(3))), *cube*(2)), denoting node ②,

mobile(*mobile*(**occ**(*cube*(1)), *cube*(3)), *cube*(2)) denoting node ③,

mobile(*mobile*(*cube*(1), **occ**(*cube*(3))), *cube*(2)) denoting node ④ and

mobile(*mobile*(*cube*(1), *cube*(3)), **occ**(*cube*(2))), denoting node ⑤.

The information (attribute) associated with an occurrence in a term is viewed as a function (called *attribute function*) applied to the associated occurrence term yielding the attribute value. An attribute function applied to an occurrence term is called *attribute occurrence*. E.g. to describe that the value of the attribute occurrence of the attribute *depth* at node ③ is the value of the attribute occurrence of the attribute *depth* at node ② plus one is denoted by

depth(*mobile*(*mobile*(**occ**(*cube*(1)), *cube*(3)), *cube*(2))) =
  depth(*mobile*(**occ**(*mobile*(*cube*(1), *cube*(3))), *cube*(2))) + 1

whereby *depth* is a function with functionality *depth*: *Mobile* → *Nat* and the sort of the occurrence term is *Mobile* as for the constructor term *mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)).

Like in algebraic specifications axioms (in this context: attribute equations) have to be defined for a set of terms and not only for one concrete term. Therefore a possible notation for an attribute equation is

depth(*mobile*(**occ**(*m*1), *m*2)) = *depth*(**occ**(*mobile*(*m*1, *m*2))) + 1

with identifiers *m*1 and *m*2 for arbitrary mobiles, denoting that with every submobile the depth is incremented.

However, in algebraic specifications identifiers are used for describing subterms. In the new approach the information to be specified is often context dependent. Therefore it would be desirable to allow identifiers describing some context in which a term appears, especially for remote access of attribute occurrences. Because of that the notion of term is extended to *scheme term* allowing *subterm identifiers*[3] being place holders for arbitrary terms with special insertion places. Other reasons for introducing subterm identifiers are: The use of subterm identifiers allows the change of the underlying grammar

[3] Such an identifier is not called *higher-order identifier* since it is viewed in a very syntactical level (cf. related work 7.2.4 and unification 5.1).

with minimal changes in the specification concerning the remote access of attributes. Defining remote access using parent-children-pathes forces to change the whole attribution when a production is inserted or deleted which influences the parent-children-pathes. Properties of attribute occurrences, which have to be proven, can be formulated using subterm identifiers. Furthermore, in the framework of deduction it is possible to describe e.g. program fragments which have to be completed. The property, which must be valid for the derived term, can be formulated using subterm identifiers. Moreover, subterm identifiers have advantages for showing the correctness of implementation relations if the attribution changes.

Let us consider the scheme term

$sv[cube(l)]$

such that $sv_{Mobile \rightarrow Mobile}$ (for short: sv) is a subterm identifier unifiable, i.e. can be made syntactically equal, with any term having an insertion place of sort *Mobile* and l is a usual identifier of sort *Mobile* (see figure 5).

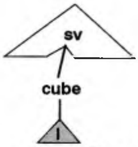


figure 5: subterm identifier and scheme term

This scheme term $sv[cube(l)]$ with the subterm identifier sv is unifiable with *mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)) with the following substitutions (figure 6 and figure 7, the insertion place is marked with the distinguished identifier z_{Mobile}):

① [$sv_{Mobile \rightarrow Mobile}$ / *mobile*(*mobile*(z_{Mobile}, *cube*(3)), *cube*(2)), l / 1],

② [$sv_{Mobile \rightarrow Mobile}$ / *mobile*(*mobile*(*cube*(1), z_{Mobile}), *cube*(2)), l / 3] and

③ [$sv_{Mobile \rightarrow Mobile}$ / *mobile*(*mobile*(*cube*(1), *cube*(3)), z_{Mobile}), l / 2].

I.e. e.g. the subterm identifier $sv_{Mobile \rightarrow Mobile}$ is replaced by *mobile*(*mobile*(z_{Mobile}, *cube*(3)), *cube*(2)) and l is replaced by 1 in case ①.

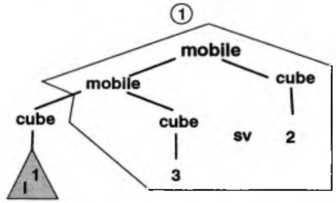


figure 6: subterm identifier and unification with a given term - part 1

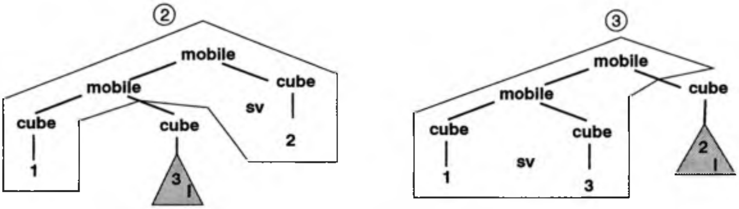


figure 7: subterm identifier and unification with a given term - part 2

This example illustrates that for scheme terms with subterm identifiers there exists no most general unifier. Nevertheless, it can be shown that a complete set of minimal unifiers can be computed. A formal definition of substitution and unification is given later.

Typical problems can be solved with the notion of a subterm identifier. E.g. in the mobile example the following attribution could be specified: The depth of each mobile is calculated from the root of the mobile to the tips (attribute: *depth*). The maximal depth of each submobile is computed from the tips to the root (attribute: *cmaxdepth*). Now the maximal depth of the mobile can be specified for each occurrence term of sort *Mobile* as the calculated maximal depth at the root of the mobile (attribute: *maxdepth*; visualized in figure 8).

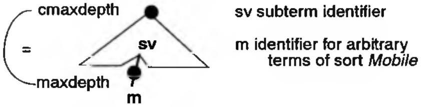


figure 8: visualized attribute equation

The corresponding attribute equation looks like:

$$maxdepth(sv[\mathbf{occ}(m)]) = cmaxdepth(\mathbf{occ}(sv[m]))$$

3.2 Syntax

It is a well known fact from [Chirica, Martin 76] that grammars can be translated into signatures viewing the abstract syntax tree as a term over a corresponding signature.

Therefore it is not dealt with grammars, but with signatures corresponding to them. Terms representing such an abstract syntax tree are ground constructor terms, i.e. usual terms built with constructor symbols, but without identifiers. Beyond ground occurrence terms other terms, like attribute occurrences, have to be built in the framework of attributed algebraic specifications as seen in the motivation.

For the definition of a scheme term the notion of a *subterm identifier* has to be characterized.

Definition 3.2.1 (subterm identifier)

A *subterm identifier* $sv_{s_1, s_2, \ldots, s_n \rightarrow s}$ (for short: *sv* and $sv_{\bar{s} \rightarrow s}$) is an identifier with a given functionality

$fct(sv_{s_1, s_2, \ldots, s_n \rightarrow s}) = s_1, s_2, \ldots, s_n \rightarrow s$

If $n = 0$ then $sv_{\rightarrow s}$ is abbreviated by sv_s, i.e. sv_s is a usual identifier.

With $SV_{s_1, s_2, \ldots, s_n \rightarrow s}$ (for short: $SV_{\bar{s} \rightarrow s}$) a set of subterm identifiers is denoted such that

for all $sv \in SV_{s_1, s_2, \ldots, s_n \rightarrow s}$ holds $fct(sv) = s_1, s_2, \ldots, s_n \rightarrow s$. ♦

Terms allowing subterm identifiers are called *scheme terms* and are defined by:

Definition 3.2.2 (scheme term)

Let $SV = (SV_{\bar{s} \rightarrow s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \rightarrow s}$ of subterm identifiers. The set of Σ-*scheme terms* (for short: *scheme terms*) of sort s with identifiers SV is denoted by $ST_\Sigma(SV)_s$ and is inductively defined by:

(1) if $t_1, t_2, \ldots, t_n$ are Σ-scheme terms of sort $s_1, s_2, \ldots, s_n$ ($n \geq 0$) and $sv \in SV_{s_1, s_2, \ldots, s_n \rightarrow s}$ is a subterm identifier, then $sv[t_1, t_2, \ldots, t_n]$ is a Σ-scheme term of sort s.
Especially each (usual) identifier $sv_s \in SV_s$ is a Σ-scheme term of sort s.

(2) if $t_1, t_2, \ldots, t_n$ are Σ-scheme terms of sort $s_1, s_2, \ldots, s_n$ and $(f\colon s_1, s_2, \ldots, s_n \rightarrow s) \in C \cup F$ ($n \geq 0$), then $f(t_1, t_2, \ldots, t_n)$ is a Σ-scheme term of sort s.
Especially each constant $(f\colon \rightarrow s) \in C \cup F$ is a Σ-scheme term of sort s.

The set of Σ-scheme terms is denoted by $ST_\Sigma(SV)_{s \in S}$ and abbreviated by $ST_\Sigma(SV)$.

Notation: Instead of $sv_{s_1, s_2, \ldots, s_n \rightarrow s}[z_{s_1}, z_{s_2}, \ldots, z_{s_n}]$ with distinguished identifiers $z_{s_1}, z_{s_2}, \ldots, z_s$ we write $sv_{s_1, s_2, \ldots, s_n \rightarrow s}$. ♦

But for pragmatic reasons the class of terms considered in this thesis is restricted, i.e. they are a proper subset of scheme terms. The signature over which the scheme terms for attributed algebraic specifications are built contains the distinguished occurrence constructor symbols, too. Thus with this term notion it is possible to construct scheme terms of the form

weight(**occ**(*mobile*(**occ**(*cube*(2)), *cube*(3))))

with no intuitive meaning. Especially on the proof theoretical side one has to consider such terms leading to a property which is for intuitive terms valid, but would not be valid for such terms. These terms can be excluded adding conditions to the properties which result in more complex properties.

Therefore the notion of term is restricted from the beginning to „intuitively well formed“ ones, called *attribute terms*, defined as:

Definition 3.2.3 (attribute term, occurrence term)

Let $\Sigma = (S, C, F)$ be a signature with a distinguished set of function symbols $F_{Attr} \subseteq F$, called *attribute function symbols*, each $f_{Attr} \in F_{Attr}$ has a functionality $f_{Attr}\colon s_{root} \rightarrow s$ for some sort s, $s_{root} \in S$. Let the function *nodesorts*: $F_{Attr} \rightarrow P(S)$ yield the sorts of the occurrences the attributes are associated with, $SV = (SV_{\bar{s} \rightarrow s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \rightarrow s}$ of subterm identifiers with $\bar{s}, s \in S^*, S$.

Moreover, let $Occ = (occ_s{:}s \rightarrow s)_{s \in \bigcup_{f_{Attr} \in F_{Attr}} nodesorts(f_{Attr})}$ be a set of distinguished constructor symbols not in C, i.e. $C \cap Occ = \varnothing$. By C_{Occ} the set $C \cup Occ$ and by Σ_{Occ} the signature $\Sigma = (S, C_{Occ}, F)$ is denoted.

The set of *attribute terms* over Σ of sort s with identifiers in SV is denoted by $AT_{\Sigma}(SV)_s$ and is inductively defined as:

(1) each scheme term $t \in ST_{(S, C, F \setminus F_{Attr})}(SV)_s$ is an attribute term of sort s.

(2) if $(f_{Attr}\colon s_1 \rightarrow s) \in F_{Attr}$ and $(f_{Attr_1}\colon s_2 \rightarrow s_1) \in F_{Attr}$ then $f_{Attr}(c[\mathbf{occ}_{s_3}(t)])$ and $f_{Attr}(\mathbf{occ}_{s_1}(f_{Attr_1}(c_1[\mathbf{occ}_{s_4}(t_1)])))$[4] are attribute terms of sort s,
with contexts $c[z_{s3}] \in ST_{(S, C, \varnothing)}(SV \cup \{ z_{s_3} \})_{s1}$, $c_1[z_{s4}] \in ST_{(S, C, \varnothing)}(SV \cup \{ z_{s4} \})_{s2}$, $t \in ST_{(S, C, \varnothing)}(SV)_{s_3}$, $t_1 \in ST_{(S, C, \varnothing)}(SV)_{s4}$, $s_3 \in nodesorts(f_{Attr})$ and $s_4 \in nodesorts(f_{Attr_1})$.
$c[\mathbf{occ}_{s_3}(t)]$ is called *occurrence term* with such c and t. The set of all occurrence terms with the distinguished occurrence constructor $\mathbf{occ}_s$ is denoted by $T^{occ_s}_{(S, C, \varnothing)}(SV)$. By $T^{Occ}_{(S, C, \varnothing)}(SV)$ the set $(T^{occ_s}_{(S, C, \varnothing)}(SV))_{s \in S}$ is denoted.

(3) if $(f\colon s_1, s_2, \ldots, s_n \rightarrow s) \in C \cup (F \setminus F_{Attr})$ and $t_1, t_2, \ldots, t_n$ are attribute terms of sort $s_1, s_2, \ldots, s_n$ $(n \geq 0)$, then $f(t_1, t_2, \ldots, t_n)$ is an attribute term of sort s.

The set of attribute terms is denoted $(AT_{\Sigma}(SV)_s)_{s \in S}$ and abbreviated by $AT_{\Sigma}(SV)$. ◆

Informally an attribute term is a term with subterm identifiers and occurrence constructor symbols, restricted to those terms formulating properties between attribute occurrences.

Note, that this definition allows to define attribute dependencies between different terms, since e.g.

$$f(f_{Attr}(c[\mathbf{occ}_s(t)]), f_{Attr_1}(c_1[\mathbf{occ}_s(t_1)]))$$

is allowed whereby the sort of context c is s and the sort of context c_1 is s_1 with different sorts s and s_1 (for some $f_{Attr}, f_{Attr_1} \in F_{Attr}, f \in F \setminus F_{Attr}$ and terms t and t_1 of appropriate sort). The attribute occurrences $f_{Attr}(c[\mathbf{occ}_s(t)])$ and $f_{Attr_1}(c_1[\mathbf{occ}_s(t_1)])$ belong to different terms since the root sorts of the occurrence terms are different. In the following it is assumed that the considered specifications describe only attribute dependencies between attribute occurrences of a single term, called *intra-attributed.*

The function *Term(t)* yields the corresponding term of a given occurrence term t and the function *OccTerms(t)* yields all occurrence terms of a given term t.

Definition 3.2.4 (*Term(t), OccTerms(t)*)

The function

$$Term\colon T^{Occ}_{(S, C, \varnothing)}(SV) \rightarrow ST_{(S, C, \varnothing)}(SV)$$

yields the corresponding term of an occurrence term and is inductively defined as

[4] This definition subsumes higher-order attribute grammars.

$Term(sv[t_1, t_2,\ldots, t_n]) = sv[Term(t_1), Term(t_2),\ldots, Term(t_n)]$,
 if $sv \in SV_{s_1, s_2,\ldots, s_n \rightarrow s}$,

$Term(f(t_1, t_2,\ldots, t_n)) = f(Term(t_1), Term(t_2),\ldots, Term(t_n))$,
 if $(f\colon s_1, s_2,\ldots, s_n \rightarrow s) \in C$,

$Term(\mathbf{occ}_s(t)) = t$

and a function

$OccTerms\colon ST_{(S, C, \varnothing)}(SV) \rightarrow P(T^{occ_s}_{(S, C, \varnothing)}(SV))$

yields the corresponding set of occurrence terms of a given term and is inductively defined as:

$OccTerms(t) = OccTermsWithContext(t, z_s)$,

$OccTermsWithContext(sv[t_1, t_2,\ldots, t_n], c[z_s]) =$
$\bigcup_{1 \le i \le n} OccTermsWithContext(t_i, c[sv[\ldots, z_{s_i},\ldots]]) \cup \{ c[\mathbf{occ}_s(sv[t_1, t_2,\ldots, t_n])] \}$,
 if $sv \in SV_{s_1, s_2,\ldots, s_n \rightarrow s}$

$OccTermsWithContext(f(t_1, t_2,\ldots, t_n), c[z_s]) =$
$\bigcup_{1 \le i \le n} OccTermsWithContext(t_i, c[f(\ldots, z_{s_i},\ldots)]) \cup \{ c[\mathbf{occ}_s(f(t_1, t_2,\ldots, t_n))] \}$,
 if $(f\colon s_1, s_2,\ldots, s_n \rightarrow s) \in C$ ♦

Because of the extension of terms the notion of substitution has to be adapted:

Definition 3.2.5 (scheme substitution, well formedness)

Let $SV = (SV_{\bar{s} \rightarrow s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \rightarrow s}$ of subterm identifiers.

A *scheme substitution* (for short: *substitution*) is a family of mappings

$\sigma = (\sigma_s\colon SV_{s_1, s_2,\ldots, s_n \rightarrow s} \rightarrow AT_\Sigma(SV \cup \{ z_{s_1}, z_{s_2},\ldots, z_{s_n} \})_s)_{s_1, s_2,\ldots, s_n, s \in S^*, S}$.

A scheme substitution $\sigma = (\sigma_s\colon SV_{s_1, s_2,\ldots, s_n \rightarrow s} \rightarrow AT_\Sigma(\{ z_{s_1}, z_{s_2},\ldots, z_{s_n} \})_s)_{s_1, s_2,\ldots, s_n, s \in S^*, S}$ is called *ground scheme substitution.*

A scheme substitution σ is denoted by

$[\, sv_1 / c_1[z_{11},\ldots, z_{1n_1}],\ldots, sv_m / c_m[z_{m1},\ldots, z_{mn_m}] \,]$ if

$dom(\sigma) = \{sv_1,\ldots, sv_m\}$.

and $\sigma(sv_1) = c_1,\ldots, \sigma(sv_m) = c_m$.

A bijective substitution σ is called *renaming scheme substitution.*

The extension of σ to attribute terms and scheme terms is denoted by σ* and defined by

(1) $\sigma^*(f(t_1, t_2,\ldots, t_n)) = f(\sigma^*(t_1), \sigma^*(t_2),\ldots, \sigma^*(t_n))$,
 with $(f\colon s_1, s_2,\ldots, s_n \rightarrow s) \in C_{Occ} \cup F$ and $t_i \in AT_\Sigma(SV)_{s_i} (1 \le i \le n)$

(2) $\sigma^*(sv[t_1, t_2,\ldots, t_n]) = c[\sigma^*(t_1), \sigma^*(t_2),\ldots, \sigma^*(t_n)]$,
with $sv \in SV_{s1, s2,\ldots, s_n \rightarrow s}$, $t_i \in AT_\Sigma(SV)_{s_i}$ with $1 \leq i \leq n$
and $\sigma(sv) = c[z_{s_1}, z_{s_2},\ldots, z_{s_n}]$,
especially $\sigma^*(sv) = \sigma(sv)$.

A substitution σ is called *well formed* for an attributed term $t \in AT_\Sigma(SV)$, if $\sigma(t) \in AT_\Sigma(SV)$.

Notation: $t\, \sigma\tau$ is used for $\tau(\sigma(t))$ with $t \in AT_\Sigma(SV)$.

Usually instead of σ^* σ is used.

σ and the notion of well formedness can be inductively extended to sets of terms and formulae.

The set of all substitutions, ground substitutions and renaming substitutions is denoted by *Subst*, *GrdSubst* and *Renaming*, respectively. ◆

In the following the well formedness of the considered substitutions is assumed.

Formulae which can be built over attribute terms are defined by:

Definition 3.2.6 (scheme formula)

Let $\Sigma = (S, C, F)$ be a signature and $SV = (SV_{\bar{s} \rightarrow s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \rightarrow s}$ of subterm identifiers. The set of (well formed) Σ-*scheme formulae* (for short: *formulae*) is inductively defined as:

(1) If $t, r \in AT_\Sigma(SV)_s$ are attribute terms of sort $s \in S$, then $t = r$ is a Σ-scheme formula (called *equation*).
(2) If Φ, Ψ are Σ-scheme formulae, then $\neg\Phi$, $\Phi \wedge \Psi$, $\Phi \vee \Psi$, $\Phi \Rightarrow \Psi$ are Σ-scheme formulae.
(3) If Φ is a Σ-scheme formula, then $\forall\, sv_{s1, s2,\ldots, s_n \rightarrow s}.\ \Phi$ and $\exists\, sv_{s1, s2,\ldots, s_n \rightarrow s}.\ \Phi$ are Σ-scheme formulae, if $sv_{s1, s2,\ldots, s_n \rightarrow s} \in SV_{s1, s2,\ldots, s_n \rightarrow s}$, especially $\forall\, sv_s.\ \Phi$ and $\exists\, sv_s.\ \Phi$ are Σ-scheme formulae. ◆

Thus we have the necessary notations to define an attributed algebraic specification.

Definition 3.2.7 (attributed algebraic specification)

An *attributed algebraic specification* is a tuple $ASpec = \langle\Sigma, F_{Attr}, Ax\rangle$ whereby

(1) $\Sigma = (S, C, F)$ is a signature consisting of a set of sort symbols S, a set of constructor symbols C and a set of function symbols F.
(2) $F_{Attr} \subseteq F$ is a set of function symbols denoting the attribute function symbols, split into inherited $F_{Attr_{inh}}$ and synthesized attribute function symbols $F_{Attr_{synth}}$ with $F_{Attr} = F_{Attr_{inh}} \cup F_{Attr_{synth}}$ and $F_{Attr_{inh}} \cap F_{Attr_{synth}} = \emptyset$.
(3) Ax is a set of axioms describing the properties of the constructor and (attribute) functions having the form $t = r$ with $t, r \in AT_\Sigma(SV)_s$ for some $s \in S$. ◆

This definition allows *undirected* attribute equations. Undirected attribute equations can result in a set of correct attributions for a given tree (see section 8.3).

Notations for Attributed Algebraic Specifications

In the following attributed algebraic specifications are named and written as

aspec specname =

sorts *S* **cons** *C* **opns** *F* **attrs synth** $F_{Attr_{synth}}$ **inh** $F_{Attr_{inh}}$ **axioms** *Ax*

endspec ♦

The axioms specify on the one side the properties of the functions as in a usual equational algebraic specification, denoted by $AlgAx(Ax) = \{\ t = r \mid t, r \in T_\Sigma(SV)_s\ \}$, and on the other side the (undirected) attribute equations, denoted by $AttrAx(Ax) = Ax \setminus AlgAx(Ax)$. Furthermore, in the axioms term/tree transformations and higher-order attribute equations can be described (see related work section 7.1.2 and 7.1.1).

Attribute grammars are a special case of attributed algebraic specifications.

If

- the algebraic specification of the functions is described in a functional way,
- only local attribute dependencies exist,
- only directed attributed equations are used and
- the usual conditions for attribute grammars hold on the attribute equations,

then a usual functional attribute grammar is obtained. Therefore the following considerations like behaviour, proving techniques, implementation relations and so on can also be applied to usual attribute grammar systems where the semantical functions are written in a functional way.

Definition 3.2.8 (functional attribute grammar)

An attributed algebraic specification is a *functional attribute grammar* iff

(1) all functions $f \in F \setminus F_{Attr}$ are constructor completely defined, i.e. can be viewed as a functional program,

(2) the attribute equations define only local attribute dependencies and are directed, i.e. the attribute equations are of the form
$f_{Attr}(sv[\mathbf{occ}(f(x_1, x_2, \ldots, x_n))]) = t$ or $f_{Attr}(sv[f(\ldots, \mathbf{occ}(x_i), \ldots)]) = t$, such that
$Occ(t) \subseteq \{\ sv[\mathbf{occ}(f(x_1, x_2, \ldots, x_n))], sv[f(\ldots, \mathbf{occ}(x_j), \ldots)]\ \}$
such that $Occ(t)$ yields the occurrence terms of the term t, $sv \in SV_{s1,\ s2,\ldots,\ s_n \rightarrow s}$, $t \in AT_\Sigma(SV)_s$ for some sort $s \in S$, $(f\colon s_1, s_2, \ldots, s_n \rightarrow s) \in C$ and $x_i \in SV_{s_i}$ with $1 \le i \le n$, and

(3) the usual conditions on attribute grammars hold, i.e. for all attribute occurrences exists a defining occurrence. ♦

Example 3.2.9

An attributed algebraic specification defining the mobile example with the described attribution looks like:

```
aspec LMOBILE =
 enrich NAT by
  sorts Mobile
  cons mobile: Mobile, Mobile → Mobile,
         cube: Nat → Mobile
  attrs synth weight, leftlength, rightlength, cmaxdepth: Mobile → Nat
         inh   length, depth: Mobile → Nat
  axioms for all sv: Mobile → Mobile; m, m1, m2: Mobile; l: Nat.
  (1)    weight(sv[occ(cube(l))]) = l,
  (2)    weight(sv[occ(mobile(m1, m2))]) =
           weight(sv[mobile(occ(m1), m2)]) + weight(sv[mobile(m1, occ(m2))]),

  (3)    length(sv[occ(mobile(m1, m2))]) =
           leftlength(sv[occ(mobile(m1, m2))]) + rightlength(sv[occ(mobile(m1, m2))]),

  (4)    weight(sv[mobile(occ(m1), m2)]) * leftlength(sv[occ(mobile(m1, m2))]) =
           weight(sv[mobile(m1, occ(m2))]) * rightlength(sv[occ(mobile(m1, m2))]),

  (5)    depth(occ(m)) = 1,
  (6)    depth(sv[mobile(occ(m1), m2)]) = depth(sv[occ(mobile(m1, m2))]) + 1,
  (7)    depth(sv[mobile(m1, occ(m2))]) = depth(sv[occ(mobile(m1, m2))]) + 1,

  (8)    cmaxdepth(sv[occ(cube(l))]) = depth(sv[occ(cube(l))]),
  (9)    cmaxdepth(sv[occ(mobile(m1, m2))]) =
           max(cmaxdepth(sv[mobile(occ(m1), m2)]), cmaxdepth(sv[mobile(m1, occ(m2))]))
endspec
```

The specification *LMOBILE* is an enrichment of the natural numbers *NAT.* A formal definition of the specification building operation *enrich* is given in section 3.5. The signature consists of the sort *Mobile* expressing a mobile with two submobiles or cubes. The constructor *mobile* takes the left and the right submobile of a floor and yields a new mobile. *cube* takes the length of cube as a natural number. The attributes are the attributes described above such that *Mobile* defines the node sort of the considered constructor terms. Axiom (1) defines the weight of a cube as its length (simplification!). Axiom (2) states that the weight of a mobile with two submobiles is the sum of the weights of the submobiles (simplification!). Axiom (3) specifies the length of a mobile as the sum of the left and the right length of this mobile from the fixing of the mobile. Axiom (4) denotes the balance equation for a mobile depending on the fixing of the mobile. The depth of a top mobile is one (axiom (5)) and for each submobile of a mobile the depth is incremented by one (axioms (6) and (7)). The maximal depth of a submobile (*cmaxdepth*) for a cube is the actual depth of the cube (axioms (8)) and for a mobile the maximum of the maximal depths of the left and right submobile (axiom (9)). ♦

3.3 Semantics

The semantics of an attributed algebraic specification consists of its signature and its model class like in standard algebraic specifications. But the attribution part has to be considered for the new specification technique, too.

The signature of an attributed algebraic specification is the given signature of the specification extended with the distinguished constructor symbols to denote the occurrences of a term.

Definition 3.3.1 (signature of an attributed algebraic specification)

Let *ASpec* = $<\Sigma, F_{Attr}, Ax>$ be an attributed algebraic specification with $\Sigma = (S, C, F)$. The signature of *ASpec* is defined as

$$sig(ASpec) = (S, C_{Occ}, F) \qquad \blacklozenge$$

Because of the new notion of formulae, a new satisfaction relation has to be defined to characterize the model class of an attributed algebraic specification as the set of *algebras* satisfying the axioms of the specification. Therefore the notion of *valuation* has to be adapted to handle subterm identifiers. This valuation can be extended to an *interpretation* for attribute terms being the unique Σ-homomorphic extension of the valuation to attribute terms. Given an algebra and a valuation each attribute term is mapped to an element of the algebra by the interpretation function.

Definition 3.3.2 (valuation, attributed interpretation)

Let $\Sigma = (S, C, F)$ be a signature, $SV = (SV_{\bar{s} \to s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \to s}$ of subterm identifiers, $A = ((A_s)_{s \in S}, (f^A)_{f \in C_{Occ} \cup F})$ be a Σ_{Occ}-algebra and $F_{Attr} \subseteq F$ be a distinguished set of attribute function symbols.

A family of mappings

$$v = (v_{s_1, s_2, \ldots, s_n \to s}: SV_{s_1, s_2, \ldots, s_n \to s} \to [A_{s_1}, A_{s_2}, \ldots, A_{s_n} \to A_s]_{term})_{s_1, s_2, \ldots, s_n, s \in S^*,}$$

is called *valuation* for *SV*. Especially for all $sv \in SV_s$ the usual valuation is obtained.

$[A_{s_1}, A_{s_2}, \ldots, A_{s_n} \to A_s]_{term}$ denotes the domain of all constructor term functions[5].

The *interpretation* of attribute terms wrt. v is a family of mappings

$$I_v^A = (I_{v,s}^A: AT_\Sigma(SV)_s \to A_s)_{s \in S}$$

defined as

(1) $I_{v,s}^A\ [sv[t_1, t_2, \ldots, t_n]] = sv^A(I_{v,s_1}^A\ [t_1], I_{v,s_2}^A\ [t_2], \ldots, I_{v,s_n}^A\ [t_n])$ such that
$sv \in SV_{s_1, s_2, \ldots, s_n \to s}$ and $sv^A = v_{s_1, s_2, \ldots, s_n \to s}(sv)$.
Especially for each $sv_s \in SV_s$ holds $I_{v,s}^A\ [sv_s] = v_s(sv_s)$.

(2) $I_{v,s}^A\ [f(t_1, t_2, \ldots, t_n)] = f^A(I_{v,s_1}^A\ [t_1], I_{v,s_2}^A\ [t_2], \ldots, I_{v,s_n}^A\ [t_n])$
with $(f: s_1, s_2, \ldots, s_n \to s) \in C_{Occ} \cup F$, $f^A: A_{s_1}, A_{s_2}, \ldots, A_{s_n} \to A_s$.

with $t_i \in AT_\Sigma(SV)_{s_i}$ $(1 \leq i \leq n)$. ♦

The interpretation of attribute terms is for given valuation v and algebra A the unique Σ-homomorphic extension of v to $AT_\Sigma(SV)$.

Lemma 3.3.3

The interpretation function I_v^A: $AT_\Sigma(SV) \to A$ is the well defined unique Σ-homomorphic

[5] Term functions are sufficient in this context, because the subterm identifiers are viewed in a syntactical way.

extension of v to attribute terms. ◆

Proof

The assertion can be shown using structural induction on the notion of attribute terms. ◆

With the notion of interpretation it is possible to define a satisfaction relation over scheme formulae used as a basis for the definition of model classes. The attributed satisfaction relation defines the validity of an equation between two attribute terms, i.e. an equation is valid if the interpretation of the two terms is the same element in the algebra and not valid otherwise. This satisfaction relation for equations can be extended to arbitrary scheme formulae.

Definition 3.3.4 (satisfaction relation)

For any Σ_{Occ}-algebra $A = ((A_s)_{s \in S}, (f^A)_{f \in C_{Occ} \cup F})$ with signature $\Sigma = (S, C, F)$, Σ-scheme formula Φ and valuation v the relation A *satisfies* Φ *wrt.* v (written $A, v \models_{\text{attr}} \Phi$) is defined by:

(1) $A, v \models_{\text{attr}} t = r$ is valid, if $I_v^A[t] = I_v^A[r]$,

(2) $A, v \models_{\text{attr}} \neg \Phi$ is valid, if $A, v \models_{\text{attr}} \Phi$ is not valid,

(3) $A, v \models_{\text{attr}} \Phi \wedge \Psi$ is valid, if $A, v \models_{\text{attr}} \Phi$ and $A, v \models_{\text{attr}} \Psi$ is valid,

(4) $A, v \models_{\text{attr}} \Phi \vee \Psi$ is valid, if $A, v \models_{\text{attr}} \Phi$ or $A, v \models_{\text{attr}} \Psi$ is valid,

(5) $A, v \models_{\text{attr}} \Phi \Rightarrow \Psi$ is valid, if $A, v \models_{\text{attr}} (\neg \Phi) \vee \Psi$ is valid,

(6) $A, v \models_{\text{attr}} \forall\, sv_{s1, s2, \ldots, s_n \to s}.\ \Phi$ is valid,
if for all valuations v' with $v(sv') = v'(sv')$ for all $sv' \in (SV_{\bar{s} \to s})_{\bar{s}, s \in S^*, S} \setminus \{ sv \}$
holds: $A, v' \models_{\text{attr}} \Phi$.

(7) $A, v \models_{\text{attr}} \exists\, sv_{s1, s2, \ldots, s_n \to s}.\ \Phi$ is valid, if $\neg(\forall\, sv_{s1, s2, \ldots, s_n \to s}.\ (\neg \Phi))$ is valid

with $t, r \in AT_\Sigma(SV)_s$ for some sort $s \in S$, Σ-scheme formulae Φ, Ψ.

A satisfies ax (written: $A \models_{\text{attr}} ax$) iff for all valuations v: $A, v \models_{\text{attr}} ax$ is valid.

A satisfies Ax (written: $A \models_{\text{attr}} Ax$) with a set of Σ-scheme formulae Ax iff for all Σ-scheme formulae $ax \in Ax$: $A \models_{\text{attr}} ax$ is valid.

$A \models_{\text{battr}} t = r$ is the abbreviation for $A \models_{\text{attr}} c[t] = c[r]$ for all observable contexts $c[z]$. ◆

Since occurrence terms are syntactical objects the reachability of the considered algebras is assumed. An algebra is reachable on a sort s with a set of constructor symbols C, if each element of the carrier set of this sort is the interpretation of some term built with constructor symbols in C, distinguished occurrence constructor symbol and subterm identifiers not of sort s.

Definition 3.3.5 (reachability)

Let $A = ((A_s)_{s \in S}, (f^A)_{f \in C_{Occ} \cup F})$ be a Σ_{Occ}-algebra with signature $\Sigma = (S, C, F)$ and $SV = (SV_{\bar{s} \to s})_{\bar{s}, s \in S^*, S}$ be a family of sets $SV_{\bar{s} \to s}$ of subterm identifiers.

An algebra A is *reachable on s with* C_1, if for each element of the carrier set $a \in A_s$ there is an attribute term $t \in AT_{(S,\, C_1 \cup Occ,\, \varnothing)}(SV')_s$ with $SV' = (SV_{\bar{s} \rightarrow s})_{\bar{s},\, s \in S^*,\, S \setminus \{s\}}$ and a valuation v with $I_v^A\,[t] = a$.

If A is reachable on all sorts $s \in S$ with C, then A is called *term generated.*

The *range* of the constructors is defined by

$range_\Sigma(C) = \{\ s \in S \mid (f\colon s_1, s_2, \ldots, s_n \rightarrow s) \in C\ \}$.

A is reachable on $range_{(S,\, C,\, F)}(C)$ with C is denoted by $A \models_{\text{attr}} C$.

Let $A \in Alg(\Sigma_2)$ and $\Sigma_1 \subseteq \Sigma_2$. $<A|_{\Sigma_1}>$ denotes the reachable Σ_1-algebra which is obtained from A by first forgetting all sort and operation symbols of Σ_2 not belonging to Σ_1 and then restricting to those elements which are reachable by the constructors of Σ_1. ♦

The semantics of an attributed algebraic specification consists of its signature extended with the distinguished occurrence constructor symbols, the attribution part being the set of attribute function symbols and the model class being the set of reachable algebras satisfying the axioms.

Definition 3.3.6 (semantics of an attributed algebraic specification)

The *semantics* of an attributed algebraic specification $ASpec = <(S, C, F), F_{Attr}, Ax>$ is defined by its

(1) *signature* $sig(ASpec) = (S, C_{Occ}, F)$,

(2) *attribution part* $attr(ASpec) = F_{Attr}$ and

(3) *model class* $Mod(ASpec) = \{\ A \in Alg(\Sigma_{Occ}) \mid A \models_{\text{attr}} C$ and $A \models_{\text{attr}} Ax\ \}$. ♦

3.4 Behavioural Attributed Algebraic Specifications

The notion of behaviour has been proven to be an adequate abstraction mechanism in the framework of algebraic specifications. In this section firstly, the background of behaviour is presented in the area of algebraic specifications and afterwards a motivation is given for applying the notion of behaviour to attribute grammars and especially to the new specification technique. The considerations on behaviour end with the discussion of the syntax and semantics of behavioural attributed algebraic specifications.

3.4.1 Background and Motivation

The idea of behaviour as an abstraction mechanism goes back to early papers on automata theory (cf. e.g. [Moore 56]). In the last ten years the notion of behaviour has attracted continuous interest in the area of algebraic specifications (cf. e.g. [Bernot, Bidoit 91; Knapik 91; Hennicker, Wirsing 93; Bidoit et al. 94, 95; Bidoit, Hennicker 94, 95]). The main application field of behavioural algebraic specifications is the definition of implementation relations and proving their correctness, since only the input/output behaviour of systems have to be considered performing implementation steps. A good overview over behavioural specifications and implementations is given in [Orejas et al. 91].

The most common notion in describing the behaviour is to distinguish a subset of the sorts as observable (e.g. [Reichel 81; Goguen, Meseguer 82; Hennicker 88; Bidoit et al. 94]). The idea is that some sorts or - in the context of programming languages - data types are observable and others are not observable. This notion was extended in [Bernot, Bidoit 91] to observable subsignatures.

Especially considering object-oriented programming languages satisfying the encapsulation principle, observability aspects can be considered. In this framework the state of objects can only be observed using methods being declared as public methods, i.e. observable methods. But the internal state or representation of an object can only be changed by method calls [Hennicker, Schmitz 96].

In the literature two main approaches for defining the semantics of behavioural algebraic specifications are distinguished.

In the first approach the observable semantics is defined by constructing the closure of the model class of a specification wrt. an observational equivalence relation on algebras (cf. e.g. [Reichel 81; Sannella, Tarlecki 85, 88; Wirsing 86]).

In the other approach a new satisfaction relation, called behavioural or observable satisfaction, is defined interpreting equations not as identities on the carrier sets but as behavioural equivalences on the elements of the carrier sets (cf. e.g. [Nivela, Orejas 88; Bernot, Bidoit 91; Hennicker 91]).

In [Bidoit et al. 94, 95] the connection of both approaches is studied.

Considering observability issues in the framework of attribute grammars is new. But in this area observability aspects are important and interesting to study, too.

Let us consider as an example the specification of a compiler using attribute grammars. Usually, the compilation process is split into several phases with one attribute grammar for each phase. In each phase more or less one attribute is interesting, e.g. in the type analysis phase the attribute storing the type of an expression and in the code generation phase the attribute containing the calculated code. Therefore attribute grammars can be viewed as behaviourally equivalent if the derived type or code is equivalent. But no distinction between e.g. optimized and unoptimized code should be made resulting in the behavioural equivalence of the attribute values (see the case study).

In the framework of user interface specification the only observable attribute could be the abstract specification of the user interface which must be visualized to the end user describing implicitly the next performable step of the user interface (see the case study).

The analogous fact holds for the use of attribute grammars in the framework of document architecture where only the layout of the text is observable and not the computation of the layout (again a case study).

3.4.2 Syntax

The compiler, the user interface and the document architecture example indicate already how a notion of behaviour has to be defined for attributed algebraic specifications:

Firstly, only a subset of the attributes is observable and the other ones are not observable.

Secondly, an observable attribute is not observable at all nodes of a tree, but at a distinguished set of nodes.

Finally, the values of the attribute occurrences are considered up to behavioural equivalence, based on the behavioural satisfaction relation of e.g. [Nivela, Orejas 88; Hennikker 91].

Example 3.4.2.1

Let us consider the above mobile example. A mobile is behaviourally equivalent to another mobile having the same outlook. But the density of the cube material may be different. I.e. the behaviour of the mobile specification is expressed by

- the set of observable attribute symbols *length*, *leftlength*, *rightlength*, *cmaxdepth* and *depth*,
- the set of observable sort symbols *Mobile* and *Nat* and
- the term representing the mobile.

But the attribute *weight* is not observable. ◆

Because of pragmatic reasons a set of attribute function symbols is distinguished, too. Auxiliary attributes can have the same sort as attributes which should be observed. Therefore observable attribute symbols are distinguished explicitly and not implicitly such that those attribute function symbols with observable result sorts are observable. Moreover, the code attribute can be an observable attribute as in the compiler example, but the value of this attribute has to be considered up to behavioural equivalence, since optimized and unoptimized code are behaviourally equivalent.

Therefore a behavioural attributed algebraic specification is an attributed algebraic specification with distinguished sets of observable sort and attribute function symbols.

Definition 3.4.2.2 (behavioural attributed algebraic specification)

A *behavioural attributed algebraic specification* is a tuple

$$ASpec = <(S, C, F), F_{Attr}, S_{Obs}, F_{AttrObs}, Ax>$$

with observable sort symbols $S_{Obs} \subseteq S$ and observable attribute function symbols

$F_{AttrObs} \subseteq F_{Attr} \subseteq F$. ◆

Notation for Behavioural Attributed Algebraic Specifications

In the following a behavioural attributed algebraic specification is written as

aspec <name> =
  sorts *S* **obs-sorts** S_{Obs} **cons** *C* **opns** *F* **attrs** F_{Attr} **obs-attrs** $F_{AttrObs}$ **axioms** *Ax*
endspec ◆

Example 3.4.2.3

With this notation the behavioural specification for *LMOBILE* of section 3.2 with observable sort symbols *Mobile* and *Nat* and observable attribute symbols *length, leftlength, rightlength*, *cmaxdepth* and *depth* can be written as:

```
aspec BEHLMOBILE =
  enrich NAT by
    sorts Mobile
    obs-sorts Mobile, Nat
    cons mobile: Mobile, Mobile → Mobile,
          cube: Nat → Mobile
    attrs synth  weight, leftlength, rightlength, cmaxdepth: Mobile → Nat
          inh    length, depth: Mobile → Nat
    obs-attrs length, leftlength, rightlength, depth, cmaxdepth
    axioms for all sv: Mobile → Mobile; m, m1, m2: Mobile; l: Nat.
    (1)   weight(sv[occ(cube(l))]) = l,
    (2)   weight(sv[occ(mobile(m1, m2))]) =
            weight(sv[mobile(occ(m1), m2)]) + weight(sv[mobile(m1, occ(m2))]),

    (3)   length(sv[occ(mobile(m1, m2))]) =
            leftlength(sv[occ(mobile(m1, m2))]) + rightlength(sv[occ(mobile(m1, m2))]),

    (4)   weight(sv[mobile(occ(m1), m2)]) * leftlength(sv[occ(mobile(m1, m2))]) =
            weight(sv[mobile(m1, occ(m2))]) * rightlength(sv[occ(mobile(m1, m2))]),
    (5)   depth(occ(m)) = 1,
    (6)   depth(sv[mobile(occ(m1), m2)]) = depth(sv[occ(mobile(m1, m2))]) + 1,
    (7)   depth(sv[mobile(m1, occ(m2))]) = depth(sv[occ(mobile(m1, m2))]) + 1,

    (8)   cmaxdepth(sv[occ(cube(l))]) = depth(sv[occ(cube(l))]),
    (9)   cmaxdepth(sv[occ(mobile(m1, m2))]) =
            max(cmaxdepth(sv[mobile(occ(m1), m2)]), cmaxdepth(sv[mobile(m1, occ(m2))])),

    (10)  length(sv[occ(cube(l))]) = 0,
    (11)  leftlength(sv[occ(cube(l))]) = 0,
    (12)  rightlength(sv[occ(cube(l))]) = 0
endspec                                                                              ♦
```

In this example all sorts are observable. In the case studies (chapter 8) we will see examples where only a subset of the sorts is observable.

3.4.3 Semantics

The intuitive notion of behavioural equivalence, shown for instance in the compiler, the document architecture and the user interface example, can be expressed as follows:

> Two attributed trees are behaviourally equivalent, if the values of the corresponding observable attribute occurrences trees are behaviourally equivalent in the sense of algebraic specifications.

Note, that behavioural equivalence is a kind of projection on the attribute occurrences.

In the non observable case a class of algebras was defined as the semantics of an attributed algebraic specification analogous to usual algebraic specifications.

For behavioural attributed algebraic specifications we define the semantics similar to [Chirica, Martin 79] as the set of solutions for the attribute values and abstract to a class of algebras afterwards.

However, the semantics is *not* defined as the set of solutions for all attribute occurrences, but as the set of solutions for all observable attribute occurrences up to behavioural equivalence in the sense of algebraic specifications.

Definition 3.4.3.1 (semantics)

The *semantics* of a beh. attributed algebraic specification $ASpec = <\Sigma, F_{Attr}, S_{Obs}, F_{AttrObs}, Ax>$ is defined by its

(1) *signature* $sig(ASpec) = \Sigma_{Occ}$,

(2) *attribution part* $attr(ASpec) = F_{Attr}$,

(3) *observable part* $obs\text{-}sorts(ASpec) = S_{Obs}$ and $obs\text{-}attrs(ASpec) = F_{AttrObs}$ and

(4) *behavioural models* $Beh(ASpec) = \{ A \in Alg(\Sigma_{Occ}) \mid A \models_{\text{attr}} C, A \models_{\text{beh}} AlgAx(Ax),$

$A \models_{\text{battr}} sol, sol \in Solutions(ASpec) \}$ such that,

$$Solutions(ASpec) = \{ \{ f_{Attr_1}(t_1) = r_1, \ldots, f_{Attr_n}(t_n) = r_n \}_t \mid f_{Attr_i} \in F_{AttrObs}, t \in T_{(S, C, \varnothing)}$$

$$Mod(ASpec) \models_{\text{attr}} f_{Attr_1}(t_1) = r_1 \wedge \ldots \wedge f_{Attr_n}(t_n) = r_n$$

$t_i \in (T^{occ_s}_{(S, C, \varnothing)})_{s \in nodesorts(f_{Attr_i})}, r_i \in T_{(S, C, \varnothing)}$ (term generated!) and

$OccTerms(t) = \{ t_1, \ldots, t_n \} \}$. ♦

An example for such a behavioural algebraic specification where the observable attribute values have to be considered up to behavioural equivalence is given in the compiler case study.

3.5 Structured Attributed Algebraic Specifications

Modularization concepts can be found as well in programming languages such as MODULA-2 [Wirth 85], C [Kernighan, Ritchie 78] and ADA [Ada 83] as in specification languages such as CLEAR [Burstall, Goguen 77, 80], OBJ2 [Futatsugi et al. 85], Maude [Meseguer, Winkler 92; Meseguer 90, 93a, 93b] and ASL [Wirsing 83, 86]. Handling complex software systems the advantages of modular construction are well known: Systematic reuse of components, separate realisation of modules, increase of understandability.

But in the area of attribute grammar systems structuring mechanisms are more or less neglected, only a few papers deal with this topic (cf. e.g. [Kastens, Waite 92; Dueck, Gormack 90; Farrow et al. 92]).

But considering again the specification of a compiler always the same problems have to be solved, e.g. type derivation, code generation for expressions and standard statements of imperative programming languages. The concrete syntax may be different, but from a more abstract point of view imperative programming languages like C, Pascal, MODULA-2,... share a common sublanguage. Having a library and specification modules dealing with such common parts of the languages and its analysis only the parser has to be adapted and the specification extended or restricted to the necessary functionality.

3.5.1 Structuring Mechanisms

We follow the structuring mechanisms of ASL presented e.g. in [Wirsing 86]. To shorten explanation the non behavioural and behavioural case are commonly treated. Therefore it is assumed for the non observable case that the specifications have no observable sort and attribute function symbols.

Definition 3.5.1.1 (structured (behavioural) attributed algebraic specifications)

The syntax and semantics of *structured (behavioural) attributed algebraic specifications* is defined inductively over the structure of the specifications:

(1) a *flat* attributed algebraic specification

sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

is a structured attributed algebraic specification if $ASpec = <(S, C, F), F_{Attr}, S_{Obs}, F_{Attr_{Obs}}, Ax>$ is a behavioural attributed algebraic specification.

Its semantics is defined by

$sig(ASpec) = (S, C_{Occ}, F)$,

$attr(ASpec) = F_{Attr}$,

$obs\text{-}sorts(ASpec) = S_{Obs}$,

$obs\text{-}attrs(ASpec) = F_{Attr_{Obs}}$,

$Mod(ASpec) = \{ A \in Alg((S, C_{Occ}, F)) \mid A \models_{\text{attr}} C \text{ and } A \models_{\text{attr}} Ax \}$ and

$Beh(ASpec) = \{ A \in Alg(\Sigma_{Occ}) \mid A \models_{\text{attr}} C, A \models_{\text{beh}} AlgAx(Ax),$

$A \models_{\text{battr}} sol, sol \in Solutions(ASpec) \}$.

(2) the sum of two structured attributed algebraic specifications $ASpec_1$ and $ASpec_2$

$ASpec = ASpec_1 + ASpec_2$

is a structured attributed algebraic specification.

The semantics is defined by

$sig(ASpec) = sig(ASpec_1) \cup sig(ASpec_2)$,

$attr(ASpec) = attr(ASpec_1) \cup attr(ASpec_2)$,

$obs\text{-}sorts(ASpec) = obs\text{-}sorts(ASpec_1) \cup obs\text{-}sorts(ASpec_2)$,

$obs\text{-}attr(ASpec) = obs\text{-}attr(ASpec_1) \cup obs\text{-}attr(ASpec_2)$,

$Mod(ASpec) = \{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid A|_{sig(ASpec_1)} \in Mod(ASpec_1)$

$\text{and } A|_{sig(ASpec_2)} \in Mod(ASpec_2) \}$ and

$Beh(ASpec) = \{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid A|_{sig(ASpec_1)} \in Beh(ASpec_1)$

$\text{and } A|_{sig(ASpec_2)} \in Beh(ASpec_2) \}$.

Note, that by definition $A|_{sig(ASpec_1)}$ is reachable.

(3) an *enrichment* of a structured attributed algebraic specification $ASpec'$

$ASpec$ = **enrich** $ASpec'$ **by**

sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

is a structured attributed algebraic specification if S is a set of sort symbols, S_{Obs} is the set of observable sort symbols with $S_{Obs} \subseteq S \cup sorts(sig(ASpec'))$, C is a set of constructor symbols, F is a set of function symbols, $F_{Attr} \subseteq F$ is a set of attribute function symbols, $F_{Attr_{Obs}}$ is a set of observable attribute function symbols with $F_{Attr_{Obs}} \subseteq F_{Attr} \cup attrs(ASpec')$ and $sig(ASpec') \cup (S, C_{Occ}, F)$ forms a signature. Ax is a set of equations $t = r$ with $t, r \in AT_{(S \cup S', C \cup C', F \cup F')}(SV)_s$ for some sort $s \in S \cup S'$ such that $sig(ASpec') = (S', C', F')$.

The enrich operator is viewed as an abbreviation for

enrich $ASpec$ **by sorts** S **obs-sorts** S_{Obs} **cons** C **opns** F

attrs F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax =

$ASpec + (sig(ASpec) \cup (S, C, F),$

$F_{Attr} \cup attrs(ASpec), S_{Obs} \cup obs\text{-}sorts(ASpec), F_{Attr_{Obs}} \cup obs\text{-}attrs(ASpec), Ax)$

if for all $f \in attrs(ASpec_1) \cap F_{Attr}$ holds: $f \in obs\text{-}attrs(ASpec_1)$ iff $f \in F_{Attr_{Obs}}$, and for all $s \in sorts(sig(ASpec_1)) \cap sorts(\Sigma)$ holds: $cons_s(sig(ASpec_1)) = cons_s(\Sigma)$ and $s \in S_{Obs}$ iff $s \in obs\text{-}sorts(ASpec_1)$.

Therefore the semantics of the enrich expression is the semantics of the sum expressions.

(4) a *renaming* of a structured attributed algebraic specification *ASpec'*

ASpec = **rename** *ASpec'* **by** σ

is a structured attributed algebraic specification, where *ASpec'* is a structured attributed algebraic specification and σ: $sig(ASpec') \rightarrow \Sigma$ is a bijective signature morphism with $\sigma(occ_s) = occ_{\sigma(s)}$ for all $s \in nodesorts(attr(ASpec'))$, called *compatible* with *Occ*.

The semantics is defined by

$sig(ASpec) = \Sigma$,

$attr(ASpec) = \sigma(attr(ASpec'))$,

$obs\text{-}sorts(ASpec) = \sigma(obs\text{-}sorts(ASpec'))$,

$obs\text{-}attrs(ASpec) = \sigma(obs\text{-}attrs(ASpec'))$,

$Mod(ASpec) = \{ A \in Alg(\Sigma) \mid A|_\sigma \in Mod(ASpec') \}$ and

$Beh(ASpec) = \{ A \in Alg(\Sigma) \mid A|_\sigma \in Beh(ASpec') \}$.

(5) an *export* of a signature $\Sigma \subseteq sig(ASpec')$ from a structured attributed algebraic specification *ASpec'*

ASpec = **export** Σ **from** *ASpec'*

is a structured attributed algebraic specification such that Σ contains all occ_s with $s \in nodesorts(attr(ASpec') \cap funcs(\Sigma))$.

The semantics is defined by

$sig(ASpec) = \Sigma$,

$attr(ASpec) = attr(ASpec') \cap funcs(\Sigma)$,

$obs\text{-}sorts(ASpec) = obs\text{-}sorts(ASpec') \cap sorts(\Sigma)$,

$obs\text{-}attr(ASpec) = obs\text{-}attr(ASpec') \cap funcs(\Sigma)$,

$Mod(ASpec) = \{ A|_\Sigma \in Alg(\Sigma) \mid A \in Mod(ASpec') \}$ and

$Beh(ASpec) = \{ A|_\Sigma \in Alg(\Sigma) \mid A \in Beh(ASpec') \}$.

3.5.2 Normalization

In this section it is shown that each structured (behavioural) attributed algebraic specification can be transformed into a special normal form as described for ASL in [Wirsing 86; Breu 89], i.e. it holds:

Each structured attributed algebraic specification can be normalized under some restrictions into the following form:

export Σ **from sorts** S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{AttrObs}$ **axioms** Ax

To show this fact the following lemma is necessary:

Lemma 3.5.2.1

It holds:

(1) $ASpec_1 + ASpec_2 = ASpec_2 + ASpec_1$

(2) $(ASpec_1 + ASpec_2) + ASpec_3 = ASpec_1 + (ASpec_2 + ASpec_3)$

(3) **rename** (**rename** $ASpec$ **by** σ_1) **by** σ_2 = **rename** $ASpec$ **by** $\sigma_1\sigma_2$

with structured attributed algebraic specification $ASpec$, $\Sigma = sig(ASpec)$, bijective signature morphisms $\sigma_1: \Sigma \rightarrow \Sigma_{\sigma 1}$ and $\sigma_2: \Sigma_{\sigma 1} \rightarrow \Sigma_{\sigma 2}$ compatible with Occ. The composition of signature morphisms is defined by $(\sigma_1\sigma_2)(x) = \sigma_2(\sigma_1(x))$ for some sort, constructor or function symbol x.

(4) $ASpec_1 + ASpec_2 =$

$(\Sigma_1 \cup \Sigma_2, F_{Attr_1} \cup F_{Attr_2}, S_{Obs_1} \cup S_{Obs_2}, F_{AttrObs_1} \cup F_{AttrObs_2}, Ax_1 \cup Ax_2)$

with $ASpec_1 = (\Sigma_1, F_{Attr_1}, S_{Obs_1}, F_{AttrObs_1}, Ax_1)$ and

$ASpec_2 = (\Sigma_2, F_{Attr_2}, S_{Obs_2}, F_{AttrObs_2}, Ax_2)$

and for all $f \in attrs(ASpec_1) \cap attrs(ASpec_2)$ holds: $f \in F_{AttrObs_1}$ iff $f \in F_{AttrObs_2}$, and for all $s \in sorts(\Sigma_1) \cap sorts(\Sigma_2)$ holds: $cons_s(\Sigma_1) = cons_s(\Sigma_2)$ and $s \in S_{Obs_1}$ iff $s \in S_{Obs_2}$.

In the behavioural case must hold additionally:

$Solutions(ASpec_1) \cup Solutions(ASpec_2) =$

$Solutions((\Sigma_1 \cup \Sigma_2, F_{Attr_1} \cup F_{Attr_2}, Ax_1 \cup Ax_2))$

(5) **export** Σ_1 **from** (**export** Σ_2 **from** $ASpec$) = **export** Σ_1 **from** $ASpec$

with $\Sigma_1 \subseteq \Sigma_2 \subseteq sig(ASpec)$

(6) (**export** Σ_1 **from** $ASpec_1$) + (**export** Σ_2 **from** $ASpec_2$) =

export ($\Sigma_1 \cup \Sigma_2$) **from** ($ASpec_1 + ASpec_2$)

$sig(ASpec_1) \supseteq \Sigma_1 \supseteq sig(ASpec_1) \cap sig(ASpec_2)$,

$sig(ASpec_2) \supseteq \Sigma_2 \supseteq sig(ASpec_1) \cap sig(ASpec_2)$

and for all $f \in attrs(ASpec_1) \cap attrs(ASpec_2)$ holds: $f \in obs\text{-}attrs(ASpec_1)$ iff $f \in obs\text{-}attr(ASpec_2)$, and for all $s \in sorts(\Sigma_1) \cap sorts(\Sigma_2)$ holds: $cons_s(\Sigma_1) = cons_s(\Sigma_2)$ and $s \in obs\text{-}sorts(ASpec_1)$ iff $s \in obs\text{-}sorts(ASpec_2)$.

(7) **rename** $ASpec$ **by** $\sigma = (\sigma(\Sigma), \sigma(F_{Attr}), \sigma(S_{Obs}), \sigma(F_{AttrObs}), \sigma(Ax))$

with $\sigma: \Sigma \rightarrow \Sigma_\sigma$ is a bijective signature morphism compatible with Occ and

$ASpec = (\Sigma, F_{Attr}, S_{Obs}, F_{AttrObs}, Ax)$

(8) **export** $sig(ASpec)$ **from** $ASpec = ASpec$

(9) **rename** (**export** Σ **from** $ASpec$) **by** σ = **export** $\sigma(\Sigma)$ **from** (**rename** $ASpec$ **by** σ')

with $\sigma: \Sigma \rightarrow \Sigma_\sigma$ being a bijective signature morphism compatible with Occ such that $sig(ASpec) \supseteq \Sigma$ and σ' is the extension of σ to $sig(ASpec)$.

Proof

(1) $Mod(ASpec_1 + ASpec_2) = \{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid$

$A|_{sig(ASpec_1)} \in Mod(ASpec_1)$ and $A|_{sig(ASpec_2)} \in Mod(ASpec_2) \} =$

$\{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid A|_{sig(ASpec_2)} \in Mod(ASpec_2)$ and

$A|_{sig(ASpec_1)} \in Mod(ASpec_1) \} = Mod(ASpec_2 + ASpec_1)$

analogous can be shown:

$Beh(ASpec_1 + ASpec_2) = Beh(ASpec_2 + ASpec_1)$

(2) analogous to (1)

(3) Let $A \in Mod(\textbf{\textit{rename}}\ (\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma_1)\ \textbf{\textit{by}}\ \sigma_2)$ iff

$A \in \{ A \in Alg(\Sigma_{\sigma 2}) \mid A|_{\sigma 2} \in Mod(\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma_1) \}$ iff

$A \in Alg(\Sigma_{\sigma 2})$ and $A|_{\sigma 2} \in \{ B \in Alg(\Sigma_{\sigma 1}) \mid B|_{\sigma 1} \in Mod(ASpec) \}$ iff

$A \in Alg(\Sigma_{\sigma 2})$ and $A|_{\sigma 2} \in Alg(\Sigma_{\sigma 1})$ and $(A|_{\sigma 2})|_{\sigma 1} \in Mod(ASpec)$ iff

$A \in Alg(\Sigma_{\sigma 2})$ and $A|_{\sigma 1\ \sigma 2} \in Mod(ASpec)$ iff

$A \in \{ A \in Alg(\Sigma_{\sigma 2}) \mid A|_{\sigma 1\ \sigma 2} \in Mod(ASpec) \} =$

$A \in Mod(\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma_1\, \sigma_2)$

analogous can be shown:

$Beh(\textbf{\textit{rename}}\ (\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma_1)\ \textbf{\textit{by}}\ \sigma_2) = Beh(\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma_1\, \sigma_2)$

(4) Let $A \in Mod(ASpec_1 + ASpec_2)$ iff

$A \in \{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid$

$A|_{sig(ASpec_1)} \in Mod(ASpec_1)$ and $A|_{sig(ASpec_2)} \in Mod(ASpec_2) \}$ iff

$A \in Alg(sig(ASpec_1) \cup sig(ASpec_2))$ and

$A|_{sig(ASpec_1)} \in Mod(ASpec_1)$ and $A|_{sig(ASpec_2)} \in Mod(ASpec_2)$ iff

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A|_{\Sigma 1} \in \{ B \in Alg(\Sigma_1) \mid B \models_{\mathrm{attr}} C_1$ and $B \models_{\mathrm{attr}} Ax_1 \}$

and

$A|_{\Sigma 2} \in \{ B \in Alg(\Sigma_2) \mid B \models_{\mathrm{attr}} C_2$ and $B \models_{\mathrm{attr}} Ax_2 \}$ iff

(since $cons_s(\Sigma_1) = cons_s(\Sigma_2)$ for all $s \in sorts(\Sigma_1) \cap sorts(\Sigma_2)$)

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A|_{\Sigma 1} \models_{\mathrm{attr}} C_1$ and $A|_{\Sigma 1} \models_{\mathrm{attr}} Ax_1$

and

$A|_{\Sigma 2} \models_{\mathrm{attr}} C_2$ and $A|_{\Sigma 2} \models_{\mathrm{attr}} Ax_2$ iff

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A \models_{\mathrm{attr}} C_1 \cup C_2$ and $A \models_{\mathrm{attr}} Ax_1 \cup Ax_2$.

iff

$A \in Mod(\Sigma_1 \cup \Sigma_2, F_{Attr_1} \cup F_{Attr_2}, Ax_1 \cup Ax_2)$

Behavioural case:

Let $A \in Beh(ASpec_1 + ASpec_2)$ iff

$A \in \{ A \in Alg(sig(ASpec_1) \cup sig(ASpec_2)) \mid$

$A|_{sig(ASpec1)} \in Beh(ASpec_1)$ and $A|_{sig(ASpec2)} \in Beh(ASpec_2) \}$ iff

$A \in Alg(sig(ASpec_1) \cup sig(ASpec_2))$ and

$A|_{sig(ASpec1)} \in Beh(ASpec_1)$ and $A|_{sig(ASpec2)} \in Beh(ASpec_2)$ iff

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A|_{\Sigma_1} \in \{ B \in Alg(\Sigma_1) \mid B \models_{\mathrm{attr}} C_1, B \models_{\mathrm{beh}} AlgAx(Ax_1),$

$B \models_{\mathrm{battr}} sol_1, sol_1 \in Solutions(ASpec_1) \}$

and

$A|_{\Sigma_2} \in \{ B \in Alg(\Sigma_2) \mid B \models_{\mathrm{attr}} C_2, B \models_{\mathrm{beh}} AlgAx(Ax_2),$

$B \models_{\mathrm{battr}} sol_2, sol_2 \in Solutions(ASpec_2) \}$ iff

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A|_{\Sigma_1} \models_{\mathrm{attr}} C_1, A|_{\Sigma_1} \models_{\mathrm{beh}} AlgAx(Ax_1), A|_{\Sigma_1} \models_{\mathrm{battr}} sol_1, sol_1 \in Solutions(ASpec_1)$

and

$A|_{\Sigma_2} \models_{\mathrm{attr}} C_2, A|_{\Sigma_2} \models_{\mathrm{beh}} AlgAx(Ax_2), A|_{\Sigma_2} \models_{\mathrm{battr}} sol_2, sol_2 \in Solutions(ASpec_2)$ iff

$A \in Alg(\Sigma_1 \cup \Sigma_2)$ and

$A \models_{\mathrm{attr}} C_1 \cup C_2, A \models_{\mathrm{beh}} AlgAx(Ax_1) \cup AlgAx(Ax_2), A \models_{\mathrm{battr}} sol,$

$sol \in Solutions(ASpec_1) \cup Solutions(ASpec_2)$ iff

(since $cons_s(\Sigma_1) = cons_s(\Sigma_2)$ for all $s \in sorts(\Sigma_1) \cap sorts(\Sigma_2)$ and the conditions for the solutions)

$A \in Beh(\Sigma_1 \cup \Sigma_2, F_{Attr_1} \cup F_{Attr_2}, Ax_1 \cup Ax_2)$

(5) Let $A \in Mod($***export*** Σ_1 ***from*** $($***export*** Σ_2 ***from*** $ASpec))$ iff

$A \in \{ A|_{\Sigma_1} \in Alg(\Sigma_1) \mid A \in Mod($***export*** Σ_2 ***from*** $ASpec) \}$ iff

$\exists\, B \in Mod($***export*** Σ_2 ***from*** $ASpec). A = B|_{\Sigma_1}$ iff

$\exists\, B \in \{ C|_{\Sigma_2} \in Alg(\Sigma_2) \mid C \in Mod(ASpec) \}. A = B|_{\Sigma_1}$ iff

$\exists\, C \in Mod(ASpec). A = C|_{\Sigma_2}|_{\Sigma_1}$ iff

$\exists\, C \in Mod(ASpec). A = C|_{\Sigma_1}$ iff

$A \in \{ A|_{\Sigma_1} \in Alg(\Sigma_1) \mid A \in Mod(ASpec) \}$ iff

$A \in Mod($***export*** Σ_1 ***from*** $ASpec)$

analogous can be shown:

$Beh($***export*** Σ_1 ***from*** $($***export*** Σ_2 ***from*** $ASpec)) = Beh($***export*** Σ_1 ***from*** $ASpec)$

(6) Let $A \in Mod($***export*** Σ_1 ***from*** $ASpec_1 +$ ***export*** Σ_2 ***from*** $ASpec_2)$ iff

$A|_{\Sigma_1} \in Mod($***export*** Σ_1 ***from*** $ASpec_1)$ and

$A|_{\Sigma_2} \in Mod(\textbf{\textit{export}}\ \Sigma_2\ \textbf{\textit{from}}\ ASpec_2)$ iff

$A|_{\Sigma_1} \in \{\, B|_{\Sigma_1} \in Alg(\Sigma_1) \mid B \in Mod(ASpec_1) \,\}$ and

$A|_{\Sigma_2} \in \{\, B|_{\Sigma_2} \in Alg(\Sigma_2) \mid B \in Mod(ASpec_2) \,\}$ iff

$(\exists\, B \in Mod(ASpec_1).\ A|_{\Sigma_1} = B|_{\Sigma_1})$ and $(\exists\, D \in Mod(ASpec_2).\ A|_{\Sigma_2} = D|_{\Sigma_2})$ iff

$(\exists\, B \in Alg(\Sigma_1).\ B \models_{\text{attr}} C_1, B \models_{\text{attr}} Ax_1$ and $A|_{\Sigma_1} = B|_{\Sigma_1})$

and $(\exists\, D \in Alg(\Sigma_2).\ D \models_{\text{attr}} C_2, D \models_{\text{attr}} Ax_2$ and $A|_{\Sigma_2} = D|_{\Sigma_2})$ iff

(because of the restrictions)

$(\exists\, E \in Alg(\Sigma_1 \cup \Sigma_2).\ E \models_{\text{attr}} C_1 \cup C_2$ and

$E \models_{\text{attr}} Ax_1 \cup Ax_2$ and $A|_{\Sigma_1} = E|_{\Sigma_1}$ and $A|_{\Sigma_2} = E|_{\Sigma_2})$ iff

$(\exists\, E \in Mod(ASpec_1 + ASpec_2).\ A|_{\Sigma_1} = E|_{\Sigma_1}$ and $A|_{\Sigma_2} = E|_{\Sigma_2})$ iff

$(\exists\, E \in Mod(ASpec_1 + ASpec_2).\ A|_{\Sigma_1 \cup \Sigma_2} = E|_{\Sigma_1 \cup \Sigma_2})$ iff

$A \in Mod(\textbf{\textit{export}}\ \Sigma_1 \cup \Sigma_2\ \textbf{\textit{from}}\ ASpec_1 + ASpec_2)$

Behavioural case:

Let $A \in Beh(\textbf{\textit{export}}\ \Sigma_1\ \textbf{\textit{from}}\ ASpec_1 + \textbf{\textit{export}}\ \Sigma_2\ \textbf{\textit{from}}\ ASpec_2)$ iff

$A|_{\Sigma_1} \in Beh(\textbf{\textit{export}}\ \Sigma_1\ \textbf{\textit{from}}\ ASpec_1)$ and

$A|_{\Sigma_2} \in Beh(\textbf{\textit{export}}\ \Sigma_2\ \textbf{\textit{from}}\ ASpec_2)$ iff

$A|_{\Sigma_1} \in \{\, B|_{\Sigma_1} \in Alg(\Sigma_1) \mid B \in Beh(ASpec_1) \,\}$ and

$A|_{\Sigma_2} \in \{\, B|_{\Sigma_2} \in Alg(\Sigma_2) \mid B \in Beh(ASpec_2) \,\}$ iff

$(\exists\, B \in Beh(ASpec_1).\ A|_{\Sigma_1} = B|_{\Sigma_1})$ and $(\exists\, D \in Beh(ASpec_2).\ A|_{\Sigma_2} = D|_{\Sigma_2})$ iff

$(\exists\, B \in Alg(\Sigma_1).\ B \models_{\text{attr}} C_1, B \models_{\text{beh}} AlgAx(Ax_1), B \models_{\text{battr}} sol_1,$

$sol_1 \in Solutions(ASpec_1)$ and $A|_{\Sigma_1} = B|_{\Sigma_1})$

and $(\exists\, D \in Alg(\Sigma_2).\ D \models_{\text{attr}} C_2, D \models_{\text{beh}} AlgAx(Ax_2), D \models_{\text{battr}} sol_2,$

$sol_2 \in Solutions(ASpec_2)$ and $A|_{\Sigma_2} = D|_{\Sigma_2})$ iff

(because of the restrictions)

$(\exists\, E \in Alg(\Sigma_1 \cup \Sigma_2).\ E \models_{\text{attr}} C_1 \cup C_2, E \models_{\text{beh}} AlgAx(Ax_1) \cup AlgAx(Ax_2), E \models_{\text{battr}} sol,$

$sol \in Solutions(ASpec_1) \cup Solutions(ASpec_2)$

and $A|_{\Sigma_1} = E|_{\Sigma_1}$ and $A|_{\Sigma_2} = E|_{\Sigma_2})$ iff

$(\exists\, E \in Beh(ASpec_1 + ASpec_2).\ A|_{\Sigma_1} = E|_{\Sigma_1}$ and $A|_{\Sigma_2} = E|_{\Sigma_2})$ iff

$(\exists\, E \in Beh(ASpec_1 + ASpec_2).\ A|_{\Sigma_1 \cup \Sigma_2} = E|_{\Sigma_1 \cup \Sigma_2})$ iff

$A \in Beh(\textbf{\textit{export}}\ \Sigma_1 \cup \Sigma_2\ \textbf{\textit{from}}\ ASpec_1 + ASpec_2)$

(7) Let $A \in Mod(\textbf{\textit{rename}}\ ASpec\ \textbf{\textit{by}}\ \sigma)$ iff

$A \in \{\, A \in Alg(\Sigma_\sigma) \mid A|_\sigma \in Mod(ASpec) \,\}$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \in Mod(ASpec)$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \in \{ B \in Alg(\Sigma) \mid B \models_{\mathrm{attr}} C$ and $B \models_{\mathrm{attr}} Ax \}$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \models_{\mathrm{attr}} C$ and $A|_\sigma \models_{\mathrm{attr}} Ax$ iff

$A \in Alg(\Sigma_\sigma)$ and $A \models_{\mathrm{attr}} \sigma(C)$ and $A \models_{\mathrm{attr}} \sigma(Ax)$ iff

$A \in Mod((\sigma(\Sigma), \sigma(F_{Attr}), \sigma(Ax)))$

Behavioural case:

Let $A \in$ *Beh(**rename** ASpec **by*** σ) iff

$A \in \{ A \in Alg(\Sigma_\sigma) \mid A|_\sigma \in Beh(ASpec) \}$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \in Beh(ASpec)$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \in \{ B \in Alg(\Sigma) \mid B \models_{\mathrm{attr}} C, B \models_{\mathrm{beh}} AlgAx(Ax), B \models_{\mathrm{battr}} sol,$
$sol \in Solutions(ASpec) \}$ iff

$A \in Alg(\Sigma_\sigma), A|_\sigma \models_{\mathrm{attr}} C, A|_\sigma \models_{\mathrm{beh}} AlgAx(Ax), A|_\sigma \models_{\mathrm{battr}} sol,$
$sol \in Solutions(ASpec)$ iff

$A \in Alg(\Sigma_\sigma), A \models_{\mathrm{attr}} \sigma(C), A \models_{\mathrm{beh}} AlgAx(\sigma(Ax)), A \models_{\mathrm{battr}} \sigma(sol),$
$sol \in Solutions((\sigma(\Sigma), \sigma(F_{Attr}), \sigma(Ax)))$ iff

$A \in Beh((\sigma(\Sigma), \sigma(F_{Attr}), \sigma(Ax)))$

(8) *Mod(**export** sig(ASpec) **from** ASpec)* =
$\{ A|_{sig(ASpec)} \in Alg(sig(ASpec)) \mid A \in Mod(ASpec) \} = Mod(ASpec)$
analogous can be shown:
*Beh(**export** sig(ASpec) **from** ASpec) = Beh(ASpec)*

(9) Let $A \in$ *Mod(**rename** (**export*** Σ ***from** ASpec) **by*** σ) iff

$A \in \{ A \in Alg(\Sigma_\sigma) \mid A|_\sigma \in Mod($**export** Σ **from** $ASpec) \}$ iff

$A \in Alg(\Sigma_\sigma)$ and $A|_\sigma \in \{ B|_\Sigma \in Alg(\Sigma) \mid B \in Mod(ASpec) \}$ iff

$\exists\, B \in Mod(ASpec).\ B|_\Sigma = A|_\sigma$ iff

$\exists\, B \in Mod(ASpec).\ B|_{\sigma'}|_{\sigma(\Sigma)} = A$ iff

$\exists\, B \in \{ A \in Alg(\Sigma_{\sigma'}) \mid A|_{\sigma'} \in Mod(ASpec) \}.\ B|_{\sigma(\Sigma)} = A$ iff

$\exists\, B \in$ *Mod(**rename** ASpec **by*** σ'). $B|_{\sigma(\Sigma)} = A$ iff

$A \in \{ A|_{\sigma(\Sigma)} \in Alg(\Sigma_\sigma) \mid A \in Mod($**rename** $ASpec$ **by** $\sigma') \}$ iff

$A \in$ *Mod(**export*** $\sigma(\Sigma)$ ***from** (**rename** ASpec **by*** σ'))

analogous can be shown:

*Beh(**rename** (**export*** Σ ***from** ASpec) **by*** σ) =
*Beh(**export*** $\sigma(\Sigma)$ ***from** (**rename** ASpec **by*** σ')) ♦

Corollary 3.5.2.2

Each structured attributed algebraic specification can be normalized into the following form:

export Σ **from sorts** S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

if the considered specification satisfies the restrictions stated in Lemma 3.5.2.1. ♦

Proof

The proof is done using structural induction on the definition of the structured attributed algebraic specifications:

(1) a *flat* attributed algebraic specification

sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

is in normal form because it can be written with lemma (8) as

export (S, C, F) **from**
  sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

(2) an *enrichment* of a structured attributed algebraic specification

enrich *ASpec* **by**
  sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

can be normalized because it is a special case of the normalizable sum operation (lemma (6)).

(3) a *renaming* of a structured attributed algebraic specification

rename *ASpec* **by** σ

can be normalized since it holds:

It can be assumed that *ASpec* is already in normal form (induction assertion), i.e. *ASpec* can be written as:

export Σ **from** $ASpec_{flat}$

with

$ASpec_{flat} \equiv$
  sorts S **obs-sorts** S_{Obs} **cons** C **opns** F **attrs** F_{Attr} **obs-attrs** $F_{Attr_{Obs}}$ **axioms** Ax

With (9) of the lemma holds:

rename (**export** Σ **from** $ASpec_{flat}$) **by** σ = **export** $\sigma(\Sigma)$ **from** (**rename** $ASpec_{flat}$ **by** σ')

derivable with (7) to

export $\sigma(\Sigma)$ **from**
  sorts $\sigma'(S)$ **obs-sorts** $\sigma'(S_{Obs})$ **cons** $\sigma'(C)$ **opns** $\sigma'(F)$
  attrs $\sigma'(F_{Attr})$ **obs-attrs** $\sigma'(F_{Attr_{Obs}})$ **axioms** $\sigma'(Ax)$

(4) an *export* of a subsignature Σ of a structured attributed algebraic specification

export Σ **from** *ASpec*

can be normalized since we can assumed the *ASpec* is already in normal form and with (5) of the lemma follows immediately the normal form.

(5) the sum of two attributed algebraic specifications

$ASpec_1 + ASpec_2$

can be normalized (under the assumption that the conditions stated in the above lemma hold) because it can be assumed that $ASpec_1$ and $ASpec_2$ are already in normal form. With (6) of the lemma follows immediately that the sum can be normalized. ♦

Remark

The condition of the behavioural case is a strong one. The critical point is the sum operation.

Having a look at the use of specification building operations, we can distinguish mainly two cases of application:

- specification building operations are used to extend loose specifications such that more informations about the functions are defined, i.e. the specifications are refined. An example is the extension of loose sets in such a way that we obtain sets which are implemented as ordered lists.
- specification building are also used for re-use purposes. In this case we have specifications in a library and combine existing specifications and extend them with additional data types to solve a complex task. But no information on the specifications of the library is add. E.g. we use the specification for deriving the type of an expression and for calculating the code of an expression and extend them such that a compiler for a complete functional programming language is obtained.

The condition on the constructors is obvious, since we usually assume with a data type special constructors from which informations over this data type can be built.

Condidering the behavioural restrictions, using the specifications in the re-use case the re-used specifications solve a special task and therefore they have a distinguished observable behaviour which need not be changed, if two specifications are merged. The same holds for refinements.

But the conditions on the solutions do not usually support the first application of specification building operations. Having a loose specification we can get on the one side more solutions (if e.g. in the loose specification no solution is obtained), and on the other side less solutions (if e.g. the refined specification excludes older solutions).

The second case can be usually supported, since the solution condition states that specifications are independent. This is the case, if e.g. the re-used specifications are constructor completely defined and the extension does not add additional axioms to the base specifications and e.g. defines itsself its function only constructor completely. I.e. combining two specifications which contain only axioms stating constructor complete definitions of functions or define only axioms on the new data type, the solution condition is satisfied.

3.6 Attributed Signature Flow Analysis

A well known analysis technique for grammars is grammar flow analysis. We extend this notion to attributed algebraic specifications to derive informations about the underlying signature and its attribution. Since in our context „signatures“ are used instead of „grammars“, we call it *attributed signature flow analysis*. Therefore the following definitions of bottom-up and top-down attributed signature flow analysis are an adaption and extension of [Möncke, Wilhelm 91; Wilhelm, Maurer 92] to our notions:

Definition 3.6.1 (bottom-up attributed signature flow analysis problem)

Let $ASpec = <(S, C, F), F_{Attr}, Ax>$ be an attributed algebraic specification. A *bottom-up attributed signature flow analysis problem* (for short: *signature flow analysis*) consists of

(1) a set of domains $D = (D_s)_{s \in S}$ with distinguished $\perp_s \in D_s$ (no information available)

(2) a set of propagation functions
$P = (p_f\colon D_{s_1}, D_{s_2},\ldots, D_{s_n} \rightarrow D_s)_{(f\colon s_1, s_2,\ldots, s_n \rightarrow s) \in C}$,

(3) a set of combination functions $(\Delta_s\colon 2^{D_s} \rightarrow D_s)_{s \in S}$, and

(4) a set of relations $(\subseteq_s: D_s, D_s \rightarrow Bool)_{s \in S}$. ♦

A (bottom-up) attributed signature flow analysis problem defines a recursive equational system:

$$SFA[s] = \Delta_s \{ p_f(SFA[s_1], SFA[s_2], \ldots, SFA[s_n]) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s) \in C \}$$

for all $s \in S$. The solution of the signature flow analysis is the solution of the recursive equational system.

Note, that the propagation functions can include informations about the attribution, too.

A simple algorithm for a bottom-up attributed signature flow analysis problem is:

The initialization is done assigning the information $\perp_s$ to all sorts:

```
procedure Init
begin
  forall s ∈S do SFA[s] := $\perp_s$ od
end
```

The real computation is done in the procedure *SFA*:

```
procedure SFA
begin
  change := true;
  while change do
    change := false;
    forall s ∈S do
        $SFA_s = \Delta_s \{ p_f(SFA[s_1], SFA[s_2], \ldots, SFA[s_n]) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s) \in C \}$
        if $SFA[s] \subseteq_s SFA_s$ and $SFA_s \subseteq_s SFA[s]$
          then nop
          else $SFA[s] := SFA_s \; \Delta_s \; SFA[s]$;
            change := true fi od od
end
```

Calling the procedure *Init* for the initialization and then the procedure *SFA* results in computing the signature flow information for each sort.

Definition 3.6.2 (top-down attributed signature flow analysis problem)

Let $ASpec = \langle (S, C, F), F_{Attr}, Ax \rangle$ be an attributed algebraic specification. A *top-down attributed signature flow analysis problem* (for short: *signature flow analysis*) consists of

(1) a set of domains $D = (D_s)_{s \in S}$ with distinguished $\perp_s \in D_s$ (no information available),

(2) a set of propagation functions
$P = ((p_{f,i}: D_s \rightarrow D_{s_i})_{i \in \{1, 2, \ldots, n\}})_{(f: s_1, s_2, \ldots, s_n \rightarrow s) \in C}$,

(3) a set of combination functions $(\Delta_s: 2^{D_s} \rightarrow D_s)_{s \in S}$,

(4) a set of relations $(\subseteq_s: D_s, D_s \rightarrow Bool)_{s \in S}$ and

(5) an initial value SFA_0 at the root: $SFA[s_{root}] = SFA_0$ with
$s_{root} \in \{ s_r \mid (f_{Attr}: s_r \rightarrow s) \in F_{Attr} \}$. ♦

A signature flow analysis problem defines a recursive equational system:

$$SFA[s] = \Delta_s \{ p_{f,i}(SFA[s']) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s') \in C, 1 \leq i \leq n, s_i = s \}$$

for all $s \in S$. The solution of the signature flow analysis is the solution of the recursive equational system.

Sometimes information depends on the complete context of a node and not only on the path from the root to that node. Therefore a new notion of signature flow analysis is defined, called *context-dependent attributed signature flow analysis*, being a combination of bottom-up and top-down signature flow analysis. The solution of the signature flow analysis problem is obtained in two phases: First a bottom-up signature flow analysis is performed. After computing the solution of this problem the result is used in a top-down signature flow analysis:

Definition 3.6.3 (context-dependent attributed signature flow analysis problem)

Let $ASpec$ = $<(S, C, F), F_{Attr}, Ax>$ be an attributed algebraic specification. A *context-dependent attributed signature flow analysis problem* (for short: *signature flow analysis*) consists of

(1) two sets of domains $D_b = (D_{b,s})_{s \in S}$ and $D_t = (D_{t,s})_{s \in S}$
with distinguished $\perp_s \in D_{b,s}$, $\perp_s \in D_{t,s}$ (no information available)

(2) two sets of propagation functions
$P_t = ((p_{f,i}: D_s \rightarrow D_{s_i})_{i \in \{1, 2, \ldots, n\}})_{(f: s_1, s_2, \ldots, s_n \rightarrow s) \in C}$,
and
$P_b = (p_f: D_{s_1}, D_{s_2}, \ldots, D_{s_n} \rightarrow D_s)_{(f: s_1, s_2, \ldots, s_n \rightarrow s) \in C}$,
such that $p_{f,i}$ can use the informations of the p_f, i.e. the solutions of
$SFA[s] = \Delta_{b,s} \{ p_f(SFA[s_1], SFA[s_2], \ldots, SFA[s_n]) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s) \in C \}$,

(3) a set of combination functions $(\Delta_{b,s}: 2^{D_s} \rightarrow D_s)_{s \in S}$ and $(\Delta_{t,s}: 2^{D_s} \rightarrow D_s)_{s \in S}$,

(4) two sets of relations $(\sqsubseteq_{b,s}: D_s, D_s \rightarrow Bool)_{s \in S}$ and $(\sqsubseteq_{t,s}: D_s, D_s \rightarrow Bool)_{s \in S}$,

(5) an initial value SFA_0 at the root: $SFA[s_{root}] = SFA_0$ with
$s_{root} \in \{ s_r \mid (f_{Attr}: s_r \rightarrow s) \in F_{Attr} \}$. ♦

A signature flow analysis problem defines a recursive equational system:

$$SFA[s] = \Delta_{t,s} \{ p_{f,i}(SFA[s']) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s') \in C, 1 \leq i \leq n, s_i = s \}$$

for all $s \in S$. The solution of the signature flow analysis is the solution of the recursive equational system.

Example 3.6.4

A realistic example for the signature flow analysis can be found in the user interface case study where the set of reachable menu-items is calculated which is an important property in the user interface verification.

Here we give again toy examples.

A bottom signature flow analysis problem of our mobile example is the calculation of all function symbols which can be found below the root.

The domain for all sort symbols is $D = \{ \mathit{mobile}, \mathit{cube}, \mathit{succ}, \mathit{zero} \}$.

The set of propagation functions is defined as
$p_{zero} = \{ \mathit{zero} \}$,
$p_{succ}(SFA[Nat]) = SFA[Nat] \cup \{ \mathit{succ} \}$,
$p_{cube}(SFA[Nat]) = SFA[Nat] \cup \{ \mathit{cube} \}$,
$p_{mobile}(SFA[Mobile], SFA[Mobile]) = SFA[Mobile] \cup \{ \mathit{mobile} \}$

The set of combination functions is the usual set union.

The set of relations is the usual set inclusion.

The result of this recursive equational system is the set $\{ \mathit{zero}, \mathit{succ}, \mathit{cube}, \mathit{mobile} \}$.

It can be analogously calculated using a top-down analysis which function symbols are above a node marked with *cube*.

The domain for all sort symbols is $D = \{ \mathit{mobile}, \mathit{cube}, \mathit{succ}, \mathit{zero} \}$.

The set of propagation functions is defined as
$p_{succ}(SFA[Nat]) = SFA[Nat] \cup \{ \mathit{succ} \}$,
$p_{cube}(SFA[Mobile]) = SFA[Mobile]$,
$p_{mobile}(SFA[Mobile]) = SFA[Mobile] \cup \{ \mathit{mobile} \}$

The set of combination functions is the usual set union.

The set of relations is the usual set inclusion.

The result of this recursive equational system is the set $\{ \mathit{mobile} \}$. ♦

Application areas of the analysis technique are e.g.
- the verification of the underlying signature, see the case study of the user interface verification, and
- use in heuristics of the proof and deduction principles, see the section about heuristics.

4 Attribute Dependencies and Attribute Evaluation

In this chapter attribute dependency relations are defined on attribute occurrences. Afterwards an attribute evaluation ordering is defined based on these relations. The attribute evaluation ordering is the starting point for generating efficient attribute evaluators.

4.1 Attribute Dependencies

In usual attribute grammar systems attribute dependencies are a basis for the attribute evaluation and for the generation of efficient attribute evaluators (cf. e.g. [Wilhelm, Maurer 92]. In these systems directed graphs describe the attribute dependencies since only directed attribute equations are used (called: dependency graphs). The following dependency graphs can be distinguished for attribute grammars with directed attribute equations [Wilhelm, Maurer 92]:

- The *local dependency graph* denotes the attribute dependencies stated in the attribution of a single production.
- The *global dependency graph* describes the dependencies for a given tree „putting together" the local dependency graphs.
- The *superior* and *subordinate characteristic dependency graph* of a non terminal describes the possible superior dependencies in all contexts, in which the non terminal may appear, and the subordinate dependencies of all subtrees with a root marked with this non terminal, respectively.

Since in our specification formalism it is dealt with *undirected* attribute equations, directed graphs cannot express the attribute dependencies. A new technique for the description of the attribute dependencies has to be developed. Moreover, the attribute equations define no ordering on the attribute occurrences. Therefore none of the known evaluation strategies can be applied in their pure form.

An undirected attribute equation is a predicate taking as arguments the attribute occurrences and stating that they reciprocally depend on each other. This fact can be expressed by a *dependency set* containing all attribute occurrences of an undirected attribute equation.

The aims are to determine statically

- the attribute evaluation ordering,
- the (non-)circularity of the attribute dependencies and
- the superior and subordinate characteristic set within the scope of reducing the search space for proofs.

The problems arising in the context of undirected attribute equations are:

- the attribute equations do not describe an ordering how the values of the attribute occurrences in this equation have to be computed and
- there may exist a set of correct attributions for a given tree.

In the following considerations the intra-term attribution of the considered attributed algebraic specifications is assumed.

The following dependency sets are distinguished for undirected attribute equations:

- the *local* and *global dependency set,*
- the *subordinate* and *superior characteristic dependency set,*

- the *characteristic dependency set,*
- the *instantiated local* and *global dependency set,*
- the *instantiated subordinate* and *superior characteristic dependency set* and
- the *instantiated characteristic dependency set,*

The explanations are based on two running examples having a one-pass and a two-pass attribution. The specification *LMOBILE* (Example 3.2.9) is a good starting point from the software engineering point of view because it is a loose specification where several design decisions are left open. But in order to get a two-pass attribution additional axioms are necessary resulting in the specification *CMOBILE*:

```
aspec CMOBILE =
  enrich NAT by
    sorts Mobile
    cons mobile: Mobile, Mobile → Mobile,
         cube: Nat → Mobile,
         fixed: → Nat
    attrs synth  weight, leftlength, rightlength, cmaxdepth: Mobile → Nat
          inh    length, depth: Mobile → Nat
    axioms for all sv: Mobile → Mobile; m, m1, m2: Mobile; l: Nat.
    (1)   weight(sv[occ(cube(l))]) = l,
    (2)   weight(sv[occ(mobile(m1, m2))]) =
             weight(sv[mobile(occ(m1), m2)]) + weight(sv[mobile(m1, occ(m2))]),

    (3)   length(sv[occ(mobile(m1, m2))]) =
             leftlength(sv[occ(mobile(m1, m2))]) + rightlength(sv[occ(mobile(m1, m2))]),

    (4)   weight(sv[mobile(occ(m1), m2)]) * leftlength(sv[occ(mobile(m1, m2))]) =
             weight(sv[mobile(m1, occ(m2))]) * rightlength(sv[occ(mobile(m1, m2))]),

    (5)   depth(occ(m)) = 1,
    (6)   depth(sv[mobile(occ(m1), m2)]) = depth(sv[occ(mobile(m1, m2))]) + 1,
    (7)   depth(sv[mobile(m1, occ(m2))]) = depth(sv[occ(mobile(m1, m2))]) + 1,

    (8)   cmaxdepth(sv[occ(cube(l))]) = depth(sv[occ(cube(l))]),
    (9)   cmaxdepth(sv[occ(mobile(m1, m2))]) =
             max(cmaxdepth(sv[mobile(occ(m1), m2)]), cmaxdepth(sv[mobile(m1, occ(m2))])),

    (10)  length(sv[occ(cube(l))]) = 0,
    (11)  leftlength(sv[occ(cube(l))]) = 0,
    (12)  rightlength(sv[occ(cube(l))]) = 0,

    (13)  length(sv[occ(mobile(m1, m2))]) =
             fixed * (cmaxdepth(occ(sv[mobile(m1, m2)])) - depth(sv[occ(mobile(m1, m2))]) + 1)
endspec
```

The axioms (1)-(9) of *CMOBILE* can be found in the specification *LMOBILE* of section 3.2. The axioms (10)-(12) state that the length, left length and right length of cubes is zero. (13) describes the length of a submobile depending on the maximal depth of the complete mobile, the actual depth of the submobile and a constant *fixed.* Note, that (13) describes a global attribute dependency. Because of this axiom a two-pass attribution with remote access is obtained if an efficient attribute evaluation should be performed. By efficient attribute evaluation we mean that only a minimum of unknown values of attribute values has to be determined using deduction (see the dynamic attribute evaluation algorithm in this chapter). A possible attribute evaluation ordering for the example term

mobile(*mobile*(*cube*(1), *cube*(3)), *cube*(2))

is shown in figure 9:

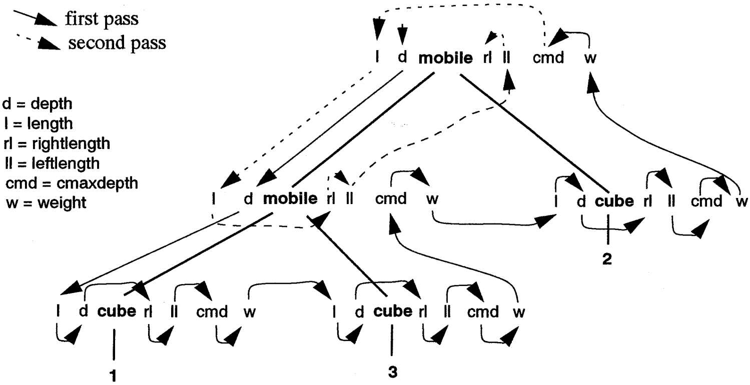


figure 9: attribute evaluation ordering for *CMOBILE*

In the first pass the depth of the nodes is calculated in a depth first tree traversal until the node *mobile*(*mobile*(**occ**(*cube*(1)), *cube*(3)), *cube*(2)) is reached. Now the value of the *weight* and *cmaxdepth* attribute occurrences can be synthesized up the tree and the values of *depth* attribute occurrences can be calculated. After the first pass the values of the *depth*, *cmaxdepth* and *weight* attribute occurrences are known. In the second pass the values of *length*, *rightlength* and *leftlength* are computed. They cannot be computed in the first pass, since the attribute occurrences *rightlength* and *leftlength* depend on the attribute occurrences *length* in turn depend on the attribute occurrences *cmaxdepth* at the root of the mobile computed as the last attribute value in the first pass.

The second attributed algebraic specification *CMOBILE*2 with a one-pass attribution is a usual attribute grammar in the new notation. The obtained results are the same as in the old dependency approaches. This examples enables a comparison of the new technique with the old one. The specification is obtained from *LMOBILE* refining the undirected balance equation (4) by directed attribute equations. Moreover, the weight of the cubes is increased. We will see later (section 6.4) that *CMOBILE*2 is a behavioural implementation of *LMOBILE*. Therefore *CMOBILE*2 is defined as a behavioural attributed algebraic specification. The observability issues have *no* influence on the attribute dependency analysis.

```
aspec CMOBILE2 =
  enrich NAT by
    sorts Mobile
    obs-sorts Mobile, Nat
    cons mobile: Mobile, Mobile → Mobile,
            cube: Nat → Mobile,
            fixed, fixedtop: → Nat
    attrs synth   weight, leftlength, rightlength, cmaxdepth: Mobile → Nat
```

```
         inh        length, depth: Mobile → Nat
      obs-attrs length, leftlength, rightlength, depth, cmaxdepth
      axioms for all sv: Mobile → Mobile; m, m1, m2: Mobile; l: Nat.
      (1)    weight(sv[occ(cube(l))]) = l * 2,
      (2)    weight(sv[occ(mobile(m1, m2))]) =
               weight(sv[mobile(occ(m1), m2)]) + weight(sv[mobile(m1, occ(m2))]),

      (3)    depth(occ(m)) = 1,
      (4)    depth(sv[mobile(occ(m1), m2)]) = depth(sv[occ(mobile(m1, m2))]) + 1,
      (5)    depth(sv[mobile(m1, occ(m2))]) = depth(sv[occ(mobile(m1, m2))]) + 1,

      (6)    cmaxdepth(sv[occ(cube(l))]) = depth(sv[occ(cube(l))]),
      (7)    cmaxdepth(sv[occ(mobile(m1, m2))]) =
               max(cmaxdepth(sv[mobile(occ(m1), m2)]), cmaxdepth(sv[mobile(m1, occ(m2))])),

      (8)    length(sv[occ(cube(l))]) = 0,
      (9)    leftlength(sv[occ(cube(l))]) = 0,
      (10)   rightlength(sv[occ(cube(l))]) = 0,

      (11)   leftlength(sv[occ(mobile(m1, m2))]) = length(sv[occ(mobile(m1, m2))]) *
               weight(sv[mobile(m1, occ(m2))]) / weight(sv[occ(mobile(m1, m2))]),
      (12)   rightlength(sv[occ(mobile(m1, m2))]) = length(sv[occ(mobile(m1, m2))]) *
               weight(sv[mobile(occ(m1), m2)]) / weight(sv[occ(mobile(m1, m2))]),

      (13)   length(occ(mobile(m1, m2))) = fixedtop,
      (14)   length(sv[mobile(occ(mobile(m1, m2)), m)]) =
               length(sv[occ(mobile(mobile(m1, m2), m))]) - fixed,
      (15)   length(sv[mobile(m, occ(mobile(m1, m2)))]) =
               length(sv[occ(mobile(m, mobile(m1, m2)))]) - fixed
  endspec
```

Axioms (1)-(7) are identical to the axioms in *LMOBILE*, with the exception that the weight of the cubes is increased in (1). (8)-(10) state that the length, left and right length of cubes is zero. The attribute equation (4) of *LMOBILE* is directed using the attribute equations (11) and (12) of *CMOBILE2* to get a usual attribute grammar. The length of the mobiles is computed in a different way to get a one-pass attribution. The length of the top mobile has a fixed value denoted by the constant *fixedtop* (axiom (13)). From a mobile to its submobiles the length is reduced by the constant *fixed* (axioms (14) and (15)).

The attribution of *CMOBILE*2 is a one-pass attribution shown in figure 10:

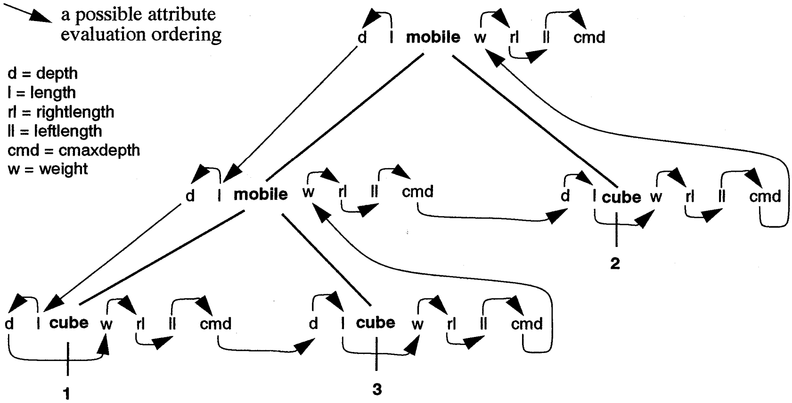


figure 10: attribute evaluation ordering for *CMOBILE*2

The attribute dependency graph for the specification *CMOBILE*2 viewed as a usual attribute grammar is given in figure 11 (note, that the arrows do not denote subordinate and superior dependencies):

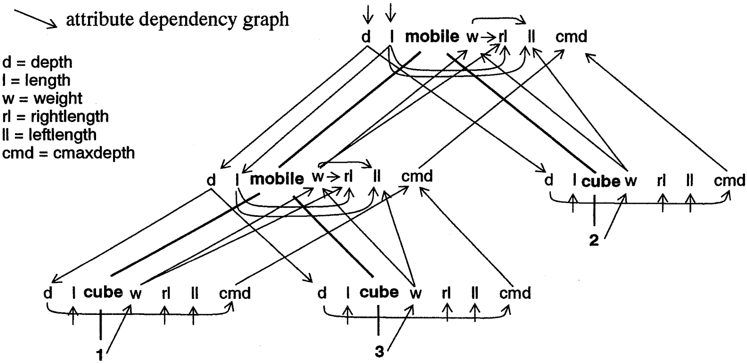


figure 11: instantiated global dependency graph

But there cannot be given a directed dependency graph for the specification *CMOBILE*, because of the undirected attribute equation:

weight(*sv*[*mobile*(**occ**(*m*1), *m*2)]) * *leftlength*(*sv*[**occ**(*mobile*(*m*1, *m*2))]) =
weight(*sv*[*mobile*(*m*1, **occ**(*m*2))]) * *rightlength*(*sv*[**occ**(*mobile*(*m*1, *m*2))])

which is an invariant of the mobile such that each floor of the mobile is in balance. In particular, each „directed“ attribute equation can be viewed as an undirected attribute equation, too.

weight(*sv*[**occ**(*mobile*(*m*1, *m*2))]) =
weight(*sv*[*mobile*(**occ**(*m*1), *m*2)]) + *weight*(*sv*[*mobile*(*m*1, **occ**(*m*2))])

can be used to calculate *weight*(*sv*[*mobile*(**occ**(*m*1), *m*2)]) knowing the values of *weight*(*sv*[**occ**(*mobile*(*m*1, *m*2))]) and *weight*(*sv*[*mobile*(*m*1, **occ**(*m*2))]), since in a logical framework - as in the new specification technique - undirected attribute equations can be solved. We obtain the following *undirected* dependency graph for the invariant and the running example term:

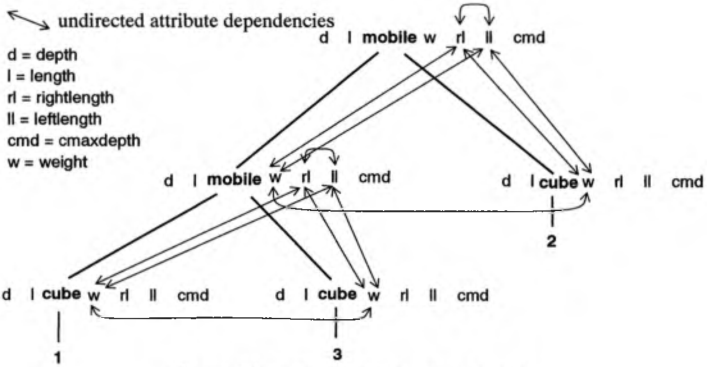


figure 12: undirected dependency graph

I.e. each attribute occurrence of the attribute equation reciprocally depends on each other. Therefore directed graphs are *not* the appropriate method for picturing the attribute dependencies.

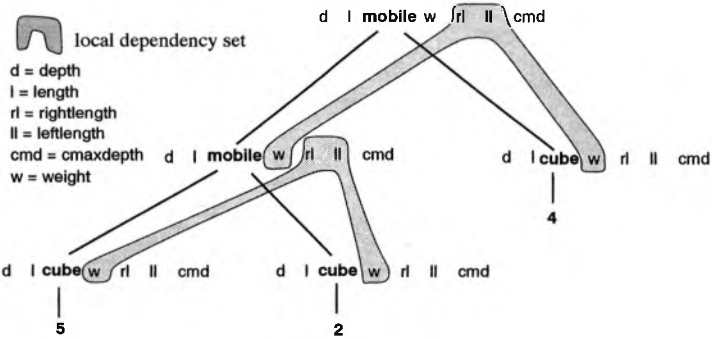


figure 13: local dependency sets

But sets describing the dependencies are more appropriate (figure 13), because they express that the attribute occurrences in an undirected attribute equation reciprocally depend on each other. We apply *dependency sets* instead of dependency graphs which are the basis for the investigation of the attribute evaluation ordering. As already mentioned several kinds of dependency sets are definable for a given specification which are investigated in detail in the following.

A *local dependency set* describes the attribute dependencies relative to an attribute equation and contains all attribute occurrences of it. This notion can be extended to sets of attribute equations.

Definition 4.1.1 (local dependency set)

The *local dependency set* for an attribute equation $t = r$ is defined as

$$DSet_{local}(t = r) = \{\, f_{Attr}(t') \mid t \equiv c[f_{Attr}(t')] \text{ or } r \equiv c[f_{Attr}(t')] \text{ for some context } c,$$
$$f_{Attr} \in F_{Attr},\ t' \in T_{\Sigma}^{occ_s}(SV) \text{ and } s \in nodesorts(f_{Attr}) \,\}$$

and for a set Ax of attribute equations as

$$DSet_{local}(Ax) = \{\, DSet_{local}(ax) \mid ax \in Ax \,\} \qquad \blacklozenge$$

Example 4.1.2

The local dependency set for *CMOBILE* is defined as (note, that the identifiers are renamed apart for later considerations):

{{ *weight*(*sv*1[**occ**(*cube*(*l*1))]) },
{ *weight*(*sv*2[**occ**(*mobile*(*m*1, *m*2))]), *weight*(*sv*2[*mobile*(**occ**(*m*1), *m*2)]),
weight(*sv*2[*mobile*(*m*1, **occ**(*m*2))]) },
{ *length*(*sv*3[**occ**(*mobile*(*m*3, *m*4))]), *leftlength*(*sv*3[**occ**(*mobile*(*m*3, *m*4))]),
rightlength(*sv*3[**occ**(*mobile*(*m*3, *m*4))]) },
{ *weight*(*sv*4[*mobile*(**occ**(*m*5), *m*6)]), *leftlength*(*sv*4[**occ**(*mobile*(*m*5, *m*6))]),
weight(*sv*4[*mobile*(*m*5, **occ**(*m*6))]), *rightlength*(*sv*4[**occ**(*mobile*(*m*5, *m*6))]) },
{ *depth*(**occ**(*m*7)) },
{ *depth*(*sv*5[*mobile*(**occ**(*m*8), *m*9)]), *depth*(*sv*5[**occ**(*mobile*(*m*8, *m*9))]) },
{ *depth*(*sv*6[*mobile*(*m*10, **occ**(*m*11))]), *depth*(*sv*6[**occ**(*mobile*(*m*10, *m*11))]) },
{ *cmaxdepth*(*sv*7[**occ**(*cube*(*l*2))]), *depth*(*sv*7[**occ**(*cube*(*l*2))]) },
{ *cmaxdepth*(*sv*8[**occ**(*mobile*(*m*12, *m*13))]),
cmaxdepth(*sv*8[*mobile*(**occ**(*m*12), *m*13)]),
cmaxdepth(*sv*8[*mobile*(*m*12, **occ**(*m*13))]) },
{ *length*(*sv*9[**occ**(*cube*(*l*3))]) },
{ *leftlength*(*sv*10[**occ**(*cube*(*l*4))]) },
{ *rightlength*(*sv*11[**occ**(*cube*(*l*5))]) },
{ *length*(*sv*12[**occ**(*mobile*(*m*14, *m*15))]), *cmaxdepth*(**occ**(*sv*12[*mobile*(*m*14, *m*15)])),
depth(*sv*12[**occ**(*mobile*(*m*14, *m*15))]) } }

and for *CMOBILE*2 as:

{{ *weight*(*sv*1[**occ**(*cube*(*l*1))]) },
{ *weight*(*sv*2[**occ**(*mobile*(*m*1, *m*2))]), *weight*(*sv*2[*mobile*(**occ**(*m*1), *m*2)]),
weight(*sv*2[*mobile*(*m*1, **occ**(*m*2))]) },
{ *length*(**occ**(*mobile*(*m*3, *m*4))) },
{ *length*(*sv*3[*mobile*(**occ**(*mobile*(*m*5, *m*6)), *m*7)]),
length(*sv*3[**occ**(*mobile*(*mobile*(*m*5, *m*6), *m*7))]) },
{ *length*(*sv*4[*mobile*(*m*8, **occ**(*mobile*(*m*9, *m*10)))]),
length(*sv*4[**occ**(*mobile*(*m*8, *mobile*(*m*9, *m*10)))]) },
{ *length*(*sv*5[**occ**(*cube*(*l*2))]) }

{ *leftlength*(*sv*6[**occ**(*mobile*(*m*11, *m*12))]), *length*(*sv*6[**occ**(*mobile*(*m*11, *m*12))]),
weight(*sv*6[*mobile*(*m*11, **occ**(*m*12))]), *weight*(*sv*6[**occ**(*mobile*(*m*11, *m*12))]) },
{ *rightlength*(*sv*7[**occ**(*mobile*(*m*13, *m*14))]), *length*(*sv*7[**occ**(*mobile*(*m*13, *m*14))]),
weight(*sv*7[*mobile*(**occ**(*m*13), *m*14)]), *weight*(*sv*7[**occ**(*mobile*(*m*13, *m*14))]) },
{ *leftlength*(*sv*8[**occ**(*cube*(*l*3))]) },
{ *rightlength*(*sv*9[**occ**(*cube*(*l*4))]) },
{ *depth*(**occ**(*m*15)) },
{ *depth*(*sv*10[*mobile*(**occ**(*m*16), *m*17)]), *depth*(*sv*10[**occ**(*mobile*(*m*16, *m*17))]) },
{ *depth*(*sv*11[*mobile*(*m*18, **occ**(*m*19))]), *depth*(*sv*11[**occ**(*mobile*(*m*18, *m*19))]) },
{ *cmaxdepth*(*sv*12[**occ**(*cube*(*l*5))]), *depth*(*sv*12[**occ**(*cube*(*l*5))]) },
{ *cmaxdepth*(*sv*13[**occ**(*mobile*(*m*20, *m*21))]),
cmaxdepth(*sv*13[*mobile*(**occ**(*m*20), *m*21)]),
cmaxdepth(*sv*13[*mobile*(*m*20, **occ**(*m*21))]) } }

which can be visualized for *CMOBILE*2 by the following dependency set:

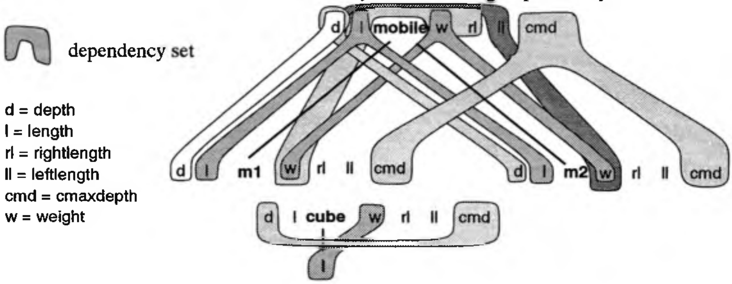


figure 14: dependency set

and the local dependency graph for attribute grammar *CMOBILE*2 is:

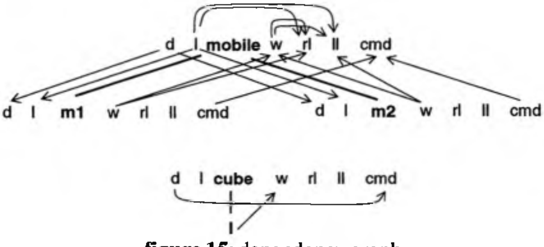


figure 15: dependency graph

The analogies between the two dependency representations strike into the eye. ♦

The local dependency set can be instantiated for a given term, i.e. the dependencies relative to the attribute equations can be calculated for the attribute occurrences of the given term. The obtained dependency set is called *instantiated local dependency set*. This notion can be extended to a set of attribute equations.

Definition 4.1.3 (instantiated local dependency set)

Given a ground constructor term t and an attribute equation ax. The *instantiated local dependency set* is defined as

$$DSet_{local}(t, ax) = \{\, \{\, \sigma(DSet_{local}(ax)) \,\} \;\Big|\; \sigma \in Subst \text{ and } \forall\, t' \in Term(ax).\ \sigma(t') \equiv t \,\}$$

and for a set of attribute equations Ax by

$$DSet_{local}(t, Ax) = \{\, DSet_{local}(t, ax) \;\Big|\; ax \in Ax \,\}$$

and

$$Inst(ax, t) = \{\, \sigma(ax) \mid \sigma \in Subst \text{ and } \forall\, t' \in Terms(ax).\ \sigma(t') \equiv t \,\}$$

such that $Terms(ax)$ is the extension of the function $Term$ to extract from an equation of attribute terms the underlying term of the syntax tree. ♦

Thus the instantiated local dependency set for all attribute equations of a specification defines *the set* of attribute dependencies for a given term.

The instantiated local dependency set for the term

$$t \equiv mobile(mobile(cube(1), cube(3)), cube(2))$$

and the specification *CMOBILE* is obtained by instantiating the above local dependency set for *CMOBILE* by the term t (see Appendix B.1).

The local dependency set is the basis for determining which values of the attribute occurrences can be calculated in parallel defining the so-called *global dependency set*. Therefore some operations on set of sets have to be defined. *Trans* defines the transitive closure of a set of sets. *UnifTrans* defines the unified transitive closure of a set of sets in such way that the elements of the set need not be identical but unifiable. The operation *DelRenamable* deletes those elements of a set of sets which are identical up to renaming of the identifiers.

Definition 4.1.4 (operations on sets)

Let $S = \{\, S_1, S_2, \ldots, S_n \,\}$ be a set of sets.

$$Trans(S) = Trans(\{\{\, \cup_{i \in I_1} S_i, \ldots, \cup_{i \in I_n} S_i \,\} \;\Big|\; 1 \le j \le n, I_j = \{\, k \;\Big|\; S_j \cap S_k \ne \varnothing, 1 \le k \le n, k \ne j \,\}\})$$

$$UnifTrans(S) = UnifTrans(\{\{\, \cup_{i \in I_1} S_i, \ldots, \cup_{i \in I_n} S_i \,\} \;\Big|\; 1 \le j \le n,$$
$$I_j = \{\, k \;\Big|\; \exists\, \sigma \in Subst.\ \sigma(S_j) \cap \sigma(S_k) \ne \varnothing, 1 \le k \le n, k \ne j \,\}\})$$

$$DelRenamable(S) = \{\, S_{i_1}, S_{i_2}, \ldots, S_{i_m} \} \text{ such that } \{\, S_{i_1}, S_{i_2}, \ldots, S_{i_m} \} \subseteq S \text{ and}$$
$$\forall\, s \in \{\, S_{i_1}, S_{i_2}, \ldots, S_{i_m} \}.\ \nexists\, s' \in S \setminus \{\, S_{i_1}, S_{i_2}, \ldots, S_{i_m} \}.\ \exists\, \sigma \in Renaming.\ \sigma(s) \equiv \sigma(s') \text{ and}$$
$$\forall\, s \in S.\ \exists\, s' \in \{\, S_{i_1}, S_{i_2}, \ldots, S_{i_m} \}.\ \exists\, \sigma \in Renaming.\ \sigma(s) \equiv \sigma(s')$$

w.l.o.g.: identifiers renamed apart! ♦

The global dependency set describes independently from the actual considered term which attribute occurrences depend on each other. The global dependency set is obtained from the local dependency set uniting those local dependency sets whose attribute occurrences are unifiable. We consider only those sets which are equal up to renaming.

Definition 4.1.5 (global dependency set)

The *global dependency set* for a set of attribute equations *Ax* is defined by

$DSet_{global}(Ax) = DelRenamable(UnifTrans(DSet_{local}(Ax)))$ ♦

Note, that in usual attribute dependency considerations global dependencies are only defined for concrete trees putting together the local dependency graphs.

The global dependency sets for the attributed algebraic specification *CMOBILE* and *CMOBILE2* are obtained from the local dependency sets given above.

The global dependency set for *CMOBILE* is:

{{ *weight*(*sv*2[**occ**(*mobile*(*m*1, *m*2))]), *weight*(*sv*2[*mobile*(**occ**(*m*1), *m*2)]),
weight(*sv*2[*mobile*(*m*1, **occ**(*m*2))]), *weight*(*sv*1[**occ**(*cube*(*l*1))]),
leftlength(*sv*4[**occ**(*mobile*(*m*5, *m*6))]), *rightlength*(*sv*4[**occ**(*mobile*(*m*5, *m*6))]),
length(*sv*3[**occ**(*mobile*(*m*3, *m*4))]), *depth*(**occ**(*m*7)),
depth(*sv*5[*mobile*(**occ**(*m*8), *m*9)]), *depth*(*sv*5[**occ**(*mobile*(*m*8, *m*9))]),
depth(*sv*6[*mobile*(*m*10, **occ**(*m*11))]), *cmaxdepth*(*sv*7[**occ**(*cube*(*l*2))]),
depth(*sv*7[**occ**(*cube*(*l*2))]), *cmaxdepth*(*sv*8[**occ**(*mobile*(*m*12, *m*13))]),
cmaxdepth(**occ**(*sv*12[*mobile*(*m*14, *m*15)])),
cmaxdepth(*sv*8[*mobile*(**occ**(*m*12), *m*13)]),
cmaxdepth(*sv*8[*mobile*(*m*12, **occ**(*m*13))]) },
{ *length*(*sv*9[**occ**(*cube*(*l*3))]) },
{ *leftlength*(*sv*10[**occ**(*cube*(*l*4))]) },
{ *rightlength*(*sv*11[**occ**(*cube*(*l*5))]) } }

and for *CMOBILE2* is:

{{ *weight*(*sv*1[**occ**(*cube*(*l*1))]), *weight*(*sv*2[**occ**(*mobile*(*m*1, *m*2))]),
weight(*sv*2[*mobile*(**occ**(*m*1), *m*2)]), *weight*(*sv*2[*mobile*(*m*1, **occ**(*m*2))]),
length(**occ**(*mobile*(*m*3, *m*4))), *length*(*sv*3[*mobile*(**occ**(*mobile*(*m*5, *m*6)), *m*7)]),
length(*sv*3[**occ**(*mobile*(*mobile*(*m*5, *m*6), *m*7))]),
length(*sv*4[*mobile*(*m*8, **occ**(*mobile*(*m*9, *m*10)))]),
leftlength(*sv*5[**occ**(*mobile*(*m*11, *m*12))]),
length(*sv*5[**occ**(*mobile*(*m*11, *m*12))]),
rightlength(*sv*6[**occ**(*mobile*(*m*13, *m*14))]),
length(*sv*6[**occ**(*mobile*(*m*13, *m*14))]) }
{ *leftlength*(*sv*7[**occ**(*cube*(*l*2)))]) },
{ *length*(*sv*7[**occ**(*cube*(*l*2))]) },
{ *rightlength*(*sv*8[**occ**(*cube*(*l*3))]) },
{ *depth*(**occ**(*m*15)), *depth*(*sv*10[*mobile*(**occ**(*m*16), *m*17)]),
depth(*sv*10[**occ**(*mobile*(*m*16, *m*17))]),
depth(*sv*11[*mobile*(*m*18, **occ**(*m*19))]), *cmaxdepth*(*sv*12[**occ**(*cube*(*l*3))]),
depth(*sv*12[**occ**(*cube*(*l*3))]), *cmaxdepth*(*sv*13[**occ**(*mobile*(*m*20, *m*21))]),
cmaxdepth(*sv*13[*mobile*(**occ**(*m*20), *m*21)]),
cmaxdepth(*sv*13[*mobile*(*m*20, **occ**(*m*21))]) } }

It can be determined for the specification *CMOBILE* that for the attribute occurrences of kind *cube* the attributes { *length* }, { *leftlength* } and { *rightlength* } and the other attribute occurrences are independently to calculate and therefore can be computed in parallel.

For the specification *CMOBILE*2 it can be extracted from the global dependency set that { *weight, length, leftlength, rightlength* } and { *depth, cmaxdepth* } can be calculated in parallel for attribute occurrences of kind *mobile* and { *length* }, { *leftlength* } and { *rightlength* } for attribute occurrences of kind *cube*. ♦

Given a term its *instantiated global dependency set* can be defined:

Definition 4.1.6 (instantiated global dependency set)

The *instantiated global dependency set* for a given term *t* and a set of attribute equations *Ax* is defined by

$DSet_{global}(t, Ax) = Trans(DSet_{local}(t, Ax))$ ♦

The instantiated global dependency set for the term *mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)) and the specifications *CMOBILE* and *CMOBILE*2 is visualized in figure 16 and figure 17, respectively.

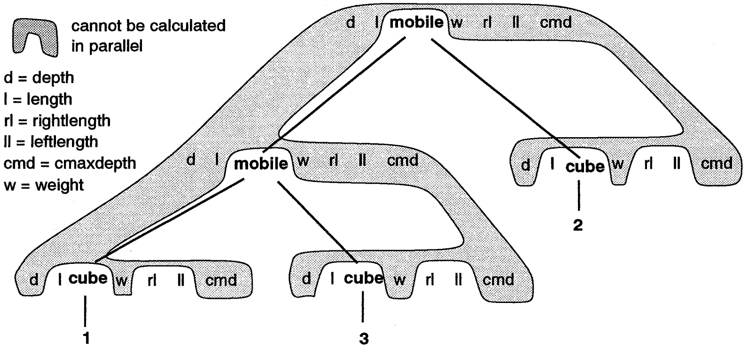


figure 16: instantiated global dependency set for *CMOBILE*

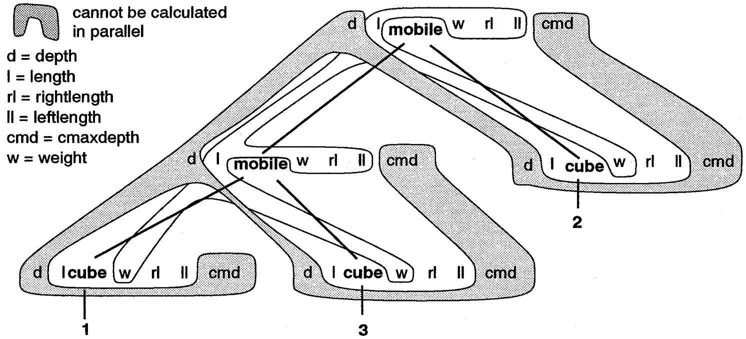


figure 17: instantiated global dependency set for *CMOBILE*2

Comparing figure 17 with the directed dependency graph in figure 11 the directed edges are changed to dependency sets.

The notion of global dependency sets is used to define the *characteristic set for occurrence terms* of a distinguished sort, describing which attribute occurrences at nodes of a distinguished sort reciprocally depend on each other. It is obtained restricting the global dependency set to attribute occurrences of the distinguished sort and abstracting to the attribute function symbols.

Definition 4.1.7 (characteristic set)

The *characteristic set for occurrence terms of sort s* and a set of attribute equations *Ax* is defined by

$$DSet_s(Ax) = Attributes(DSet_{global}(Ax) \cap \{\, f_{Attr}(t) \mid f_{Attr} \in F_{Attr},\ t \in T^{occ_s}_{(S,\,C,\,\varnothing)}(SV) \text{ and } s \in nodesorts(f_{Attr}) \,\})$$

such that $Attributes(\{\, f_{Attr_1}(t_1), f_{Attr_2}(t_2), \ldots, f_{Attr_n}(t_n) \,\}) = \{\, f_{Attr_1}, f_{Attr_2}, \ldots, f_{Attr_n} \,\}$ ♦

For the specification *CMOBILE* the characteristic set for occurrence terms of sort *Mobile* is

$$\{\, \{ weight,\ leftlength,\ rightlength,\ length,\ depth,\ cmaxdepth \,\} \,\}$$

and for the specification *CMOBILE*2 the characteristic set for occurrence terms of sort *Mobile* is

$$\{\{\, weight,\ length,\ leftlength,\ rightlength \},\ \{\, depth,\ cmaxdepth \,\}\}$$

The *instantiated characteristic set* for an occurrence term $t \equiv c[\mathbf{occ}_s(t')]$ is obtained from the global dependency set of $c[t']$ restricting it to the attribute occurrences of *t*.

Definition 4.1.8 (instantiated characteristic dependency set)

The *instantiated characteristic dependency set* for an occurrence term $t \equiv c[\mathbf{occ}_s(t')]$ and a set of attribute equations *Ax* is defined as

$$ChDSet(t, Ax) = Attributes(DSet_{global}(c[t'], Ax) \cap \{\, f_{Attr}(t) \mid f_{Attr} \in F_{Attr} \text{ and } s \in nodesorts(f_{Attr}) \,\})$$ ♦

Let $t = mobile(\mathbf{occ}(mobile(cube(1), cube(3))), cube(2))$. The characteristic dependency set for this occurrence term in the specification *CMOBILE* is

$$ChDSet(t) = \{\, \{\, weight(t),\ leftlength(t),\ rightlength(t),\ length(t) \,\},\ \{\, depth(t),\ cmaxdepth(t)\} \,\}$$

which is in this case the characteristic set of the sort *Mobile*.

But for the occurrence term $t = mobile(mobile(\mathbf{occ}(cube(1)), cube(3)), cube(2))$ the instantiated characteristic set is

$$ChDSet(t) = \{\, \{\, weight(t) \,\},\ \{\, leftlength(t) \,\},\ \{\, rightlength(t)\},\ \{\, length(t) \,\},\ \{\, depth(t),\ cmaxdepth(t)\} \,\}$$

i.e. the characteristic set is a worst case approximation. More detailed estimations are obtained using the instantiation of the equations for the characteristic terms[6] instead of the attribute equations.

In considerations about the attribute dependencies for attribute grammars the notion of subordinate and superior characteristic graph is defined at this passage.

The subordinate and superior characteristic graph for a sort describes independent of a given occurrence term the attribute dependencies in all subtrees of this sort and the attribute dependencies in all contexts with an insertion place of this sort, respectively. But in the framework of undirected attribute equations the attribute evaluation ordering has to be known to determine the corresponding subordinate and superior characteristic set.

Let us consider the term *cube*(1) and the undirected attribute equation

$$\mathit{cmaxdepth}(\mathbf{occ}(\mathit{cube}(1))) = \mathit{depth}(\mathbf{occ}(\mathit{cube}(l)))$$

In this example it is possible that either *cmaxdepth* depends on *depth*, i.e. there is a dependency in the subordinate tree, or *depth* depends on *cmaxdepth*, i.e. there is a dependency in the superior tree, depending whether *depth* or *cmaxdepth* is determined first. With the attribute equation

$$\mathit{depth}(\mathbf{occ}(\mathit{cube}(1))) = 1$$

the attribute value of *depth* can be calculated and therefore *cmaxdepth* depends on *depth*, i.e. there is a subordinate attribute dependency.

The example shows, that the subordinate and superior characteristic set, can be defined after the evaluation ordering on the attribute occurrences is determined. Because of this fact the definition of the subordinate and superior characteristic set is delayed until the attribute evaluation ordering was treated.

4.2 Attribute Evaluation

In this subsection an introduction into the attribute evaluation for usual attribute grammar systems is given. As a next step a dynamic attribute evaluation algorithm for undirected attribute equations is investigated. An attribute evaluation ordering is developed to generate attribute evaluators for the new approach. This ordering is the starting point for the definition of visit sequences on the attribute occurrences and the subordinate and superior characteristic sets.

4.2.1 Introduction to Attribute Evaluation

The following considerations are a summary of [Wilhelm, Maurer 92] for attribute grammars.

Usually two phases of the attribute evaluation are distinguished, namely the *strategy* and the *evaluation phase*:

In the *strategy phase* the attribute evaluation ordering, i.e. the ordering in which the values of the attribute occurrences have to be calculated, is derived. Several proceedings in this phase can be differentiated:

[6] The set of characteristic terms for a set of attribute equations is obtained by iterative calculating the unifying terms for these equations (see Definition 4.2.4.1).

- *Dynamic determination*:
  The individual dependency graph *DGraph*(*t*) is sorted topologically. If a total ordering can be obtained then the attribution is cycle free. This ordering is the evaluation ordering. But the technique is space and time consuming.
- *Induced evaluation ordering*:
  If for all non terminals X there exists a total ordering T_X on its attribute occurrences, such that for arbitrary trees t holds:
  If the attribute occurrences a_n is in relation with b_n at any node n in the instantiated dependency graph of t, and the node n is marked with the non terminal X, then a has to be in relation with b wrt. T_X.
  In this case T_X is called the *induced evaluation ordering*.
  Based on the total ordering a visit sequence on the attribute occurrences can be defined.
- *Selection between visit sequences*:
  If there is no induced evaluation ordering there are productions rules with different visit sequences in different contexts. In this case all possible visit sequences have to be generated for each production. The correct visit sequence is chosen during the evaluation time.

In the *evaluation phase* the values of the attribute occurrences of a given tree are calculated using the attribute evaluation ordering of the strategy phase. Two kinds are distinguishable:

- *demand driven*:
  The evaluator is called with a set of attribute occurrences, whose values have to be computed.
- *data driven*:
  The evaluator starts with the calculation of those values of attribute occurrences which depend on no other attribute occurrences. Afterwards it computes those values of attribute occurrences for which the values of the attribute occurrences, which are necessary for the calculation, are already determined.

4.2.2 Dynamic Attribute Evaluation for Undirected Attribute Equations

In the dynamic attribute evaluation the evaluation ordering is derived for a given term and not for a class of terms. Therefore the individual dependency graph *DGraph*(*t*) is sorted topologically. If a total ordering can be obtained the attribution is cycle free and can be used as an evaluation ordering. This is the proceeding for usual attribute grammars with directed attribute equations and non remote access.

4.2.2.1 Abstract Algorithm

Since in the new approach undirected attribute equations with remote access of attribute values are supported a different point of view has to be taken.

In usual dynamic attribute evaluation systems the instantiated dependency graph is topologically sorted. In the new approach it is not dealt with dependency graphs but with dependency sets, i.e. no ordering on the attribute occurrences is defined. Therefore a sorting on sets has to be developed. Moreover in attribute grammar systems, under the assumption that an attribute evaluation ordering can be defined, in each step exactly *one value* of an attribute occurrence is calculated. But in the new specification technique loose specifications are possible. E.g. an invariant between several attributes can result in a *set of values* for one attribute occurrence. Moreover, the attribution can be defined in such a way, that two or more values of attribute occurrences must be simultaneously calculated. In this case existentially quantified formulae have to be solved.

The starting point for the dynamic attribute evaluation is the instantiated local dependency set of a term relative to a set of attribute equations. The smallest attribute occurrences are the elements of the local dependency sets with cardinality one. Having already determined an ordering on some attribute occurrences O each element of O is smaller than an element of a set of attribute occurrences O' iff there is a ds and no ds' in the local dependency set such that $| ds' \cap O | < | ds \cap O |$ and $O' = ds \setminus O$. I.e. O' is the smallest set of attribute occurrences which have to be calculated simultaneously. A data-driven algorithm for the attribute evaluation ordering looks like:

procedure *dynamic_attribute_evaluation_ordering*

Input:
- term t
- set of attribute equations Ax

Output:
- $<_{eval}$ attribute evaluation ordering

```
begin
  (1)  ordered = $\{ a \mid \{ a \} \in DSet_{local}(t, Ax) \}$
  (2)  minimal elements wrt. $<_{eval}$ are all elements in ordered
  (3)  repeat
  (4)    nextdset = $\{ ds \mid ds \in DSet_{local}(t, Ax)\ \nexists\ ds' \in DSet_{local}(t, Ax).$
                    $| ds' \cap ordered | < | ds \cap ordered | \}$
  (5)    for all d $\in$ordered do
  (6)      for all a $\in$nextdset do
  (7)        d $<_{eval}$ a \ ordered
  (8)    od od
  (9)    ordered = ordered $\cup \bigcup_{ds \in nextdset} ds$
  (10) until ordered = $\bigcup_{ds \in DSet_{local}(t, Ax)} ds$
  (11) return($<_{eval}$)
end
```

The input is the term t for which the attribute evaluation ordering has to be determined and a set of attribute equations Ax. The output is the evaluation ordering $<_{eval}$ of the attribute occurrences.

In (1) the sets of the instantiated dependency sets with cardinality one are defined as the smallest elements in $<_{eval}$. Therefore the ordering of these attribute occurrences is already known (2). (3)-(10) performs a loop until all attribute occurrences are ordered. (4) calculates the dependency sets of the instantiated local dependency set with minimal unordered attribute occurrences which have to be considered next. All the already ordered attribute occurrences are smaller than the unordered attribute occurrences of the considered dependency set (5)-(8). These attribute occurrences are ordered in the following iterations (9). (11) returns the calculated attribute evaluation ordering for the input term.

Example 4.2.2.1.1

Let us determine as an example the attribute evaluation ordering for *CMOBILE* and the term:

mobile(*mobile*(*cube*(1), *cube*(3)), *cube*(2))

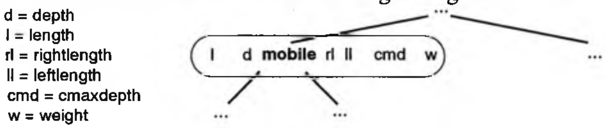
The algorithm is executed as follows:

```
(1)  ordered = { weight(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                 weight(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                 weight(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                 depth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                 length(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                 length(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                 length(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                 leftlength(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                 leftlength(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                 leftlength(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                 rightlength(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                 rightlength(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                 rightlength(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) }
(2)  the elements of ordered in (1) are the minimal elements of $<_{eval}$
(4)  nextdset = {{ weight(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                   weight(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                   weight(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },
                 { depth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                   depth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) },
                 { depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                   depth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ weight(mobile(occ(mobile(cube(1), cube(3))), cube(2)))
       d $<_{eval}$ depth(mobile(mobile(cube(1), cube(3)), occ(cube(2))))
       d $<_{eval}$ depth(mobile(occ(mobile(cube(1), cube(3))), cube(2)))
(9)  ordered = ordered ∪ {  weight(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                            depth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                            depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))) }
(4)  nextdset = {{ weight(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                   weight(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                   weight(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },
                 { depth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                   depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))) },
                 { depth(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                   depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))) },
                 { cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
                   depth(mobile(mobile(cube(1), cube(3)), occ(cube(2))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ weight(occ(mobile(mobile(cube(1), cube(3)), cube(2))))
       d $<_{eval}$ depth(mobile(mobile(occ(cube(1)), cube(3)), cube(2)))
       d $<_{eval}$ depth(mobile(mobile(cube(1), occ(cube(3))), cube(2)))
       d $<_{eval}$ cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2))))
(9)  ordered = ordered ∪ {weight(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                          depth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                          depth(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                          cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) }
(4)  nextdset = {{ cmaxdepth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                  depth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))) },
                { cmaxdepth(mobile(mobile(cube(1), occ(cube(3)), cube(2))),
                  depth(mobile(mobile(cube(1), occ(cube(3)), cube(2))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ cmaxdepth(mobile(mobile(occ(cube(1)), cube(3)), cube(2)))
       d $<_{eval}$ cmaxdepth(mobile(mobile(cube(1), occ(cube(3)), cube(2)))
(9)  ordered = ordered ∪ {cmaxdepth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                           cmaxdepth(mobile(mobile(cube(1), occ(cube(3)), cube(2))) }
(4)  nextdset = { cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  cmaxdepth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                  cmaxdepth(mobile(mobile(cube(1), occ(cube(3))), cube(2))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2)))
(9)  ordered = ordered ∪ {cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2)))}
(4)  nextdset = {{ cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                  cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2))))
(9)  ordered = ordered ∪ { cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) }
(4)  nextdset = {{ length(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                  cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2))))),
                  depth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) },
                { length(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2))),
                  depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ length(occ(mobile(mobile(cube(1), cube(3)), cube(2))))
       d $<_{eval}$ length(mobile(occ(mobile(cube(1), cube(3))), cube(2)))
(9)  ordered = ordered ∪ {length(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                           length(mobile(occ(mobile(cube(1), cube(3))), cube(2))) }
(4)  nextdset = {{ length(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                  leftlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                  rightlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) },
                { length(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  leftlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  rightlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))) },
                { weight(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  leftlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                  weight(mobile(mobile(cube(1), cube(3)), occ(cube(2))))
                  rightlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) },
                { weight(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
                  leftlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                  weight(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
                  rightlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))) } }
(5)  for all d ∈ordered do
(7)    d $<_{eval}$ { leftlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                 rightlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) }
       d $<_{eval}$ { leftlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                 rightlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))) }
(9)  ordered = ordered ∪ {leftlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                           rightlength(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
                           leftlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
                           rightlength(mobile(occ(mobile(cube(1), cube(3))), cube(2))) }   ◆
```

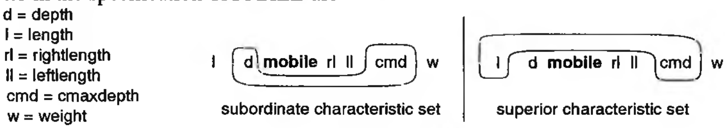
4.2.3 Attribute Evaluation Ordering

In the previous section a method was developed for dynamically determining the attribute evaluation ordering in the framework of attributed algebraic specifications. Having an algebraic specification such that the existentially quantified formulae can be solved in a good time and space complexity, e.g. having a constructor complete definition of the functions, the dynamic attribute evaluation ordering slows down the attribute evaluation. Therefore it would be desirable, especially if an attributed algebraic specification is a functional attribute grammar, to generate attribute evaluators with nearly the same time and space complexity as in the old approaches.

The subordinate characteristic set for occurrence terms of a distinguished sort defines the global dependencies of the synthesized attributes in terms of the inherited attributes, like the superior characteristic set which describes the global dependencies in the context. Thus the attribute evaluator has perfect strategical information. E.g. if the attribute evaluator visits the visualized occurrence in the following cutting of an attributed tree:



and the subordinate and superior characteristic set - without the local dependencies - for *mobile* in the specification *CMOBILE* are



subordinate characteristic set

superior characteristic set

and the value of the attribute occurrence of attribute *d* is already calculated, then the evaluator knows, that

- after visiting the subtree under the marked node the value of *cmd* is known,
- but the value of *l* is surely not known, since it depends contextually on *cmd*.

I.e. visiting this node a next time (after having the values of *d* and *cmd* calculated) the value of *l* can be computed. But as already mentioned the subordinate and superior characteristic set can only be determined knowing the attribute evaluation ordering.

Thus the proceeding performed in attribute grammar systems is not usable in the case of undirected attribute equations. A new strategy has to be investigated.

Before defining a formal strategy, it is explained exemplarily.

The basis for determining the attribute evaluation ordering is the set of characteristic terms of the attribute equations being for the specification *CMOBILE*

$\{ sv[mobile(sv1[cube(l1)], sv2[cube(l2)])] \}$.

It can be visualized as:

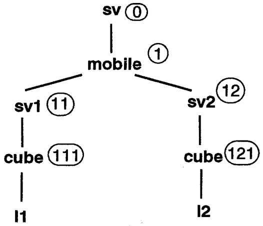


figure 18: characteristic term wrt. the attribute equations

To shorten notation the attribute occurrences are denoted by the attribute function symbols and the circled numbers in the tree of figure 18, e.g. weight_{111} denotes the attribute occurrence *weight*(*sv*[*mobile*(*sv*1[**occ**(*cube*(*l*1))], *sv*2[*cube*(*l*2)])]).
For the characteristic term the instantiated local attribute dependency set (wrt. the specification *CMOBILE*) is calculated:

$$\{\,\{\mathit{weight}_{111}\},\{\mathit{weight}_{121}\},\{\mathit{weight}_1,\mathit{weight}_{11},\mathit{weight}_{12}\},$$
$$\{\mathit{length}_1,\mathit{leftlength}_1,\mathit{rightlength}_1\},\{\mathit{weight}_{11},\mathit{leftlength}_1,\mathit{weight}_{12},\mathit{rightlength}_1\},$$
$$\{\mathit{depth}_0\},\{\mathit{depth}_1,\mathit{depth}_{11}\},\{\mathit{depth}_1,\mathit{depth}_{12}\},\{\mathit{cmaxdepth}_{111},\mathit{depth}_{111}\},$$
$$\{\mathit{cmaxdepth}_{121},\mathit{depth}_{121}\},\{\mathit{cmaxdepth}_1,\mathit{cmaxdepth}_{11},\mathit{cmaxdepth}_{12}\},$$
$$\{\mathit{leftlength}_{111}\},\{\mathit{leftlength}_{121}\},\{\mathit{rightlength}_{111}\},\{\mathit{rightlength}_{121}\},\{\mathit{length}_{111}\},$$
$$\{\mathit{length}_{121}\},\{\mathit{length}_1,\mathit{cmaxdepth}_0,\mathit{depth}_1\}\,\}.$$

We start with the sets containing only one element, since the values of the attribute occurrences can be determined independently of the others and are therefore the smallest elements in the attribute evaluation ordering:

$$\{\,\{\mathit{weight}_{111}\},\{\mathit{weight}_{121}\},\{\mathit{depth}_0\},\{\mathit{leftlength}_{111}\},\{\mathit{leftlength}_{121}\},$$
$$\{\mathit{rightlength}_{111}\},\{\mathit{rightlength}_{121}\},\{\mathit{length}_{111}\},\{\mathit{length}_{121}\}\,\}.$$

Now an ordering can be defined on the other elements analogous to the dynamic attribute evaluation ordering idea:

$$\left.\begin{array}{l}\mathit{depth}_0\le\mathit{depth}_1\le\mathit{depth}_{11}\le\mathit{depth}_{111}\le\mathit{cmaxdepth}_{111}\le\mathit{cmaxdepth}_{11}\\ \mathit{depth}_0\le\mathit{depth}_1\le\mathit{depth}_{12}\le\mathit{depth}_{121}\le\mathit{cmaxdepth}_{121}\le\mathit{cmaxdepth}_{12}\end{array}\right\}\le\mathit{cmaxdepth}_1$$

$$\mathit{cmaxdepth}_1\le\mathit{cmaxdepth}_0$$

$$\left.\begin{array}{c}\mathit{cmaxdepth}_0\\ \mathit{depth}_1\end{array}\right\}\le\mathit{length}_1\le\mathit{length}_0$$

$$\left.\begin{array}{l}\mathit{weight}_{111}\le\mathit{weight}_{11}\\ \mathit{weight}_{121}\le\mathit{weight}_{12}\end{array}\right\}\le\mathit{weight}_1\le\mathit{weight}_0$$

$$\mathit{length}_1\le\{\mathit{leftlength}_1,\mathit{rightlength}_1\}$$

$$\left.\begin{array}{l}\mathit{weight}_{11}\\ \mathit{weight}_{12}\end{array}\right\}\le\{\mathit{leftlength}_1,\mathit{rightlength}_1\}$$

The ordering $\mathit{depth}_0\le\mathit{depth}_1$, $\mathit{depth}_{11}\le\mathit{depth}_{111}$,... is valid because *depth* is an inherited attribute and using the trivial substitution for the subterm identifier results in the applicability of a dependency set (inductive definition of the ordering). The same fact holds for the synthesized attributes, e.g. $cmaxdepth_{111} \leq cmaxdepth_{11}$.

Based on this ordering a total ordering can be constructed analogous to usual attribute grammar systems. Defining an ordering taking tree traverses into consideration, e.g. the following total ordering can be constructed:

$depth_0 < depth_1 < depth_{11} < depth_{111} < weight_{111} < cmaxdepth_{111} < length_{111} < leftlength_{111}$
$< rightlength_{111} < cmaxdepth_{11} < weight_{11} < depth_{12} \leq depth_{121} < weight_{121}$
$< cmaxdepth_{121} < length_{121} < leftlength_{121} < rightlength_{121} < cmaxdepth_{12} < weight_{12}$
$< weight_1 < cmaxdepth_1 < weight_0 < cmaxdepth_0 < length_1 < length_0$
$< \{ leftlength_1, rightlength_1 \} < \{ leftlength_0, rightlength_0 \}$

I.e. especially for the nodes marked with *mobile* we get the ordering

$depth_1 < weight_1 < cmaxdepth_1 < length_1 < \{ leftlength_1, rightlength_1 \}$

and for the node marked with *cube* we get

$depth_{111} < length_{111} < weight_{111} < cmaxdepth_{111} < leftlength_{111} < rightlength_{111}$

and

$depth_{121} < length_{121} < weight_{121} < cmaxdepth_{121} < leftlength_{121} < rightlength_{121}$

Since *depth* and *length* are inherited attributes and *weight*, *cmaxdepth*, *leftlength* and *rightlength* are synthesized attributes the following visit-sequence is obtained:

In the first pass the inherited attribute *depth* is computed at nodes marked with *mobile*. After visiting the subtree the synthesized attributes *weight* and *cmaxdepth* are known. In the next tree traversing the inherited attribute *length* can be determined and after visiting the subtree *leftlength* and *rightlength* are known.
Nodes marked with *cube* have to be visited once, because the inherited attributes are all calculated before the synthesized ones.

We adapt the notion of ordered partition and visit sequence of [Wilhelm, Maurer 92] to our approach.

Definition 4.2.3.1 (ordered partition)

Let < be a total attribute evaluation ordering on attribute occurrences of sort s. An *ordered partition* for < is a sequence of the form

$i^1 s^1 i^2 s^2 \ldots i^k s^k$ with $(1 \leq j \leq k)$

- $i^j \equiv f_{Attr_{j,1}}, f_{Attr_{j,2}}, \ldots, f_{Attr_{j,n_j}}$
  such that $f_{Attr_{j,l}} \in F_{Attr_{inh}} (1 \leq l \leq n_j)$, $f_{Attr_{j,1}} < f_{Attr_{j,2}}, \ldots, f_{Attr_{j,n_j-1}} < f_{Attr_{j,n_j}}$ and $i^j \not\equiv \varepsilon$ for $1 < j \leq k$.
- $s^j \equiv f_{Attr_{j,1}}, f_{Attr_{j,2}}, \ldots, f_{Attr_{j,m_j}}$
  such that $f_{Attr_{j,l}} \in F_{Attr_{synth}} (1 \leq l \leq m_j)$, $f_{Attr_{j,1}} < f_{Attr_{j,2}}, \ldots, f_{Attr_{j,m_j-1}} < f_{Attr_{j,m_j}}$ and $s^j \not\equiv \varepsilon$ for $1 \leq j < k$.

i^j is the j-th *down pass visit*, s^j is the j-th *up pass visit* and $i^j s^j$ the j-th *visit*. ◆

Definition 4.2.3.2 (visit sequence)

Let < be a total attribute evaluation ordering on the attribute occurrences of occurrence terms of sort s and

$$i_j^1\ s_j^1\ i_j^2\ s_j^2\ \dots\ i_j^{kj}\ s_j^{kj}\ (1 \leq j \leq n)$$

be the ordered partition for $(f\colon s_1, s_2, \dots, s_n \rightarrow s) \in C$ and constructor terms $t_1, t_2, \dots, t_n$ of sort $s_1, s_2, \dots, s_n$.

A *visit sequence vseq* for f and $t_1, t_2, \dots, t_n$ is an evaluation ordering of the following form:

$$vseq(sv[f(t_1, t_2, \dots, t_n)]) = i_0^1\ d_1\ s_0^1\ i_0^2\ d_2\ s_0^2\ \dots\ i_0^k\ d_k\ s_0^k$$

and

d_l is a sequence of visits $i_j^l\ s_j^l\ (1 \leq j \leq n)$ at $sv[f(\dots, \mathbf{occ}(t_j), \dots)]$. ◆

The attribute evaluation ordering can be defined updating the dynamic attribute evaluation algorithm in the way shown by the example.

procedure *attribute_evaluation_ordering*

Input:
- one characteristic term t
- set of attribute equations Ax

Output:
- $<_{eval}$ attribute evaluation ordering for the characteristic term t

```
begin
    ordered = $\{ a \mid \{ a \} \in DSet_{local}(t, Ax) \}$
    minimal elements wrt. $<_{eval}$ are all elements in ordered
    repeat
        nextdset = $\{ ds \mid ds \in DSet_{local}(t, Ax)\ \nexists\ ds' \in DSet_{local}(t, Ax).$
                        $| \sigma(ds') \cap ordered | < | \sigma(ds) \cap ordered |$
                        for some trivial subterm substitution $\sigma \}$
        for all d $\in$ordered do
            for all a $\in$nextdset do
                d $<_{eval}$ a $\setminus \cup_\sigma\, \sigma(ordered)$
        od od
        ordered = ordered $\cup \bigcup_{ds \in nextdset} ds$
    until ordered = $\bigcup_{ds \in DSet_{local}(t, Ax)} ds$
    return($<_{eval}$)
end
```

A loose specification can result in an incomplete ordering, e.g. if no defining equation for an attribute occurrence is given.

The obtained partial ordering can be normalized such that a visit sequence can be calculated.

The subordinate and superior characteristic set can be determined with the knowledge of the attribute evaluation ordering.

4.2.4 Subordinate and Superior Characteristic Set

The subordinate and superior characteristic set for a sort *s* describe independent of a given occurrence term the attribute dependencies in all subtrees of sort *s* and the attribute dependencies in all contexts with insertion places of sort *s*, respectively. The basis for both characteristic sets is the local dependency set and the attribute evaluation ordering.

Definition 4.2.4.1 (subordinate/superior characteristic set for sort s)

The *subordinate and superior characteristic set for a sort s* is calculated in 4 steps:

- The set of characteristic terms is calculated wrt. the attribute equations with $T = Terms(AttrAx(Ax))$ (under the assumption that the identifiers used in the axioms are renamed apart):

  $$Term^{char}(T) = \begin{cases} T, \text{ if } mguSet(T) = \emptyset \\ Term^{char}(\{\sigma(t) \mid \sigma \in mguSet(t), t \in T\}), \text{ otherwise} \end{cases}$$

- The characteristic local dependency set is computed wrt. the characteristic set of scheme terms.

  $DSet_{local}^{char}(Ax) \quad = \{ DSet_{local}(t, Ax) \mid t \in Term^{char}(Terms(AttrAx(Ax))) \}$

- The set of characteristic occurrence terms is calculated wrt. the distinguished sort.

  $OccTerm_s^{char}(Ax) = \{ t \mid t \in OccTerms(ct) \cap \{ c[\mathbf{occ}_s(f(t_1, t_2, \ldots, t_n))]$ for some constructor context c, constructor symbol f and $t_1, t_2, \ldots, t_n$ terms of appropriate sort $\}$, $ct \in Term^{char}(Ax) \}$

- The *subordinate characteristic set for a sort s* is defined by

  $DSet_{sub}^{s} = Attr(\cup_{t \in OccTerm_s^{char}(Ax)} (UnifTrans(D_{sub} \cap$
  $\{ t' \mid t' <_{\text{char}} t, t' \in OccTerm_s^{char}(Ax) \}) \cap \{ f_{Attr}(t) \text{ with } f_{Attr} \in F_{Attr} \}))$

  with $t_1 <_{\text{char}} t_2$, iff t_1 has the form $c[\mathbf{occ}(c'[t])]$ for some contexts c, c' and some term t and t_2 has the form $c[c'[\mathbf{occ}(t)]]$ (with $DSet = DSet_{local}^{char}(Ax)$):

  $$\text{Such that } D_{sub} = \bigcup_{d \in DSet} \begin{cases} d, \text{ if } d \text{ contains only inherited, synthesized} \\ \qquad \text{attributes, respectively} \\ \{d_1, d_2, \ldots, d_n\}, \text{ if the visit sequence of } d \text{ is} \\ \qquad i^1 s^1 i^2 s^2 \ldots i^k s^k \text{ then } d_j = \{i^j s^j\} \text{ with } 1 \le j \le k \end{cases}$$

  The *superior characteristic set for a sort s* is defined by

  $DSet_{sup}^{s} = Attr(\cup_{t \in OccTerm_s^{char}(Ax)} (UnifTrans(D_{sup} \cap$
  $\{ t' \mid \text{it does not hold: } t' <_{\text{char}} t, t' \in OccTerm_s^{char}(Ax) \} \cap$
  $\{ f_{Attr}(t) \text{ with } f_{Attr} \in F_{Attr} \}))$

  with $t_1 <_{\text{char}} t_2$, as in the subordinate characteristic set case.
  Such that

  $$D_{sup} = \bigcup_{d \in DSet} \begin{cases} d, \text{ if } d \text{ contains only inherited, synthesized} \\ \qquad \text{attributes, respectively} \\ \{d_1, d_2, \ldots, d_n\}, \text{ if the visit sequence of } d \text{ is} \\ \qquad i^1 s^1 i^2 s^2 \ldots i^k s^k \text{ then } d_j = \{s^j i^{j+1}\} \text{ with } 1 \le j < k \end{cases}$$ ◆

Example 4.2.4.2

The subordinate and superior characteristic set for the specification *CMOBILE* is computed in four steps:

- The characteristic set of scheme terms is calculated wrt. the attribute equations. $Term^{char}(Terms(AttrAx(Ax))) = \{ sv[mobile(sv1[cube(l1)], sv2[cube(l2)])] \}$
- The characteristic local dependency set is computed wrt. the characteristic set of scheme terms (using the numeration of figure 18.): $DSet_{local}^{char}(Ax) = \{ \{ weight_{111} \}, \{ weight_{121} \}, \{ weight_1, weight_{11}, weight_{12} \}, \{ length_1, leftlength_1, rightlength_1 \}, \{ weight_{11}, leftlength_1, weight_{12}, rightlength_1 \}, \{ depth_0 \}, \{ depth_1, depth_{11} \}, \{ depth_1, depth_{12} \}, \{ cmaxdepth_{111}, depth_{111} \}, \{ cmaxdepth_{121}, depth_{121} \}, \{ cmaxdepth_1, cmaxdepth_{11}, cmaxdepth_{12} \}, \{ leftlength_{111} \}, \{ leftlength_{121} \}, \{ rightlength_{111} \}, \{ rightlength_{121} \}, \{ length_{111} \}, \{ length_{121} \}, \{ length_1, cmaxdepth_0, depth_1 \} \}$.
- The set of characteristic occurrence terms is calculated wrt. the distinguished sort. $OccTerm_s^{char}(Ax) = OccTerms(sv[mobile(sv1[cube(l1)], sv2[cube(l2)])])$,
- $DSet_{sub}^{Mobile} = \{ \{ weight, length, leftlength, rightlength \}, \{ depth, cmaxdepth \} \}$
- $DSet_{sup}^{Mobile} = \{ \{ cmaxdepth, weight, length \} \}$ ♦

Remark

In our considerations we have dealt with the worst case approximation whereby all equations of the form

$$f_{Attr}(t) = rhs$$

(with attribute occurrence $f_{Attr}(t)$ and arbitrary attribute term *rhs*) are undirected attribute equations. But the proceeding can be simplified, if these equations are handled like directed attribute equations. In this case the usual ordering can be used on the attribute occurrences of these equations.

Generating Efficient Attribute Evaluators

The generation of efficient attribute evaluators can be performed like in usual attribute grammars using the visit sequences or the superior and subordinate characteristic sets.

Attribute Dependencies and Structured Attributed Algebraic Specifications

Using the structuring mechanisms presented in section 3.5 not only the specifications but also the attribute dependencies for these specifications should be stored in libraries. Some considerations have to be made to get correct attribute dependency relations for the combined specifications:

- Flat specifications do not influence the determined attribute dependencies.
- In the case of an enrichment of an attributed algebraic specification the following cases have to be differentiated:
  + The attribute dependencies can be merged, if in the axioms of the enrich-part attributes are used which are not defined or used in the basic specification, otherwise
  + a new attribute dependency analysis has to be performed for the normalized specification.
- The rename operation causes no problems, since only the signature morphism has to be

applied to the dependency sets and visit sequences.

- The export of a subsignature does not influence the attribute dependencies.
- The sum operation causes the same problems as the enrich operation:
  - if the specifications do not share common attributes then the attribute dependencies can be merged and
  - otherwise a new attribute dependency analysis has to be started for the normalized specification.

5 Calculi for Attributed Algebraic Specifications

This chapter deals with three kinds of calculi: one for determining the complete set of minimal unifiers for a set of attribute terms, one for dealing with universally quantified formulae including induction principles and a narrowing calculus for dealing with existentially quantified formulae.

5.1 Unification Calculus

Note the fundamental difference to the work on the topic of unification done in the framework of higher-order algebraic specifications, see e.g. [Heering et al. 94].

Unification solves the problem of making two terms syntactic equal, i.e. a substitution is computed, which applied to both terms, results in two syntactic equal terms, if the two terms are unifiable. Unification was first discussed in [Herbrand 30] and an algorithmic form of the computation was given in [Robinson 65]. In [Martelli, Montanari 82] unification was described by a set of rules manipulating a set of equations to obtain a most general unifier.

Since in attributed algebraic specifications the notion of terms was extended to attribute terms admitting subterm identifiers, the unification algorithm has to be adapted to handle such identifiers, too.

The following considerations are based on [Hofbauer, Kutsche 89] extended to manage subterm identifiers.

Definition 5.1.1 (unifier)

A *unifier* for two attribute terms t_1 and t_2 is a scheme substitution σ such that $\sigma(t_1) \equiv \sigma(t_2)$.

A unifier σ is more general than a unifier σ' (written: $\sigma \leq \sigma'$), iff there exists a substitution $\sigma'' \in Subst$ such that $\sigma\sigma'' = \sigma'$.

A unifier for a set of attribute terms $\{t_1, t_2, \ldots, t_n\}$ is a scheme substitution σ such that $\sigma(t_1) \equiv \sigma(t_2) \equiv \ldots \equiv \sigma(t_{n-1}) \equiv \sigma(t_n)$. ◆

Let us consider the attribute terms

$t = weight(sv1[mobile(sv2[mobile(\mathbf{occ}(m1), m2)], m3)])$ and

$r = weight(sv3[mobile(sv4[mobile(mobile(mobile(\mathbf{occ}(m4), m5), m6), m7)], m8)])$

E.g. (using the usual abbreviations, i.e. neglecting the functionality of the subterm identifiers)

$$\begin{aligned}\sigma = [\ & sv1_{Mobile \rightarrow Mobile} \,/\, sv3_{Mobile \rightarrow Mobile},\\ & sv2_{Mobile \rightarrow Mobile} \,/\, sv4[mobile(mobile(z_{Mobile}, m6), m7)],\\ & m1 \,/\, m4, m2 \,/\, m5, m3 \,/\, m8\ ] \text{ and}\end{aligned}$$

$$\begin{aligned}\sigma' = [\ & sv1_{Mobile \rightarrow Mobile} \,/\, sv3[mobile(sv4_{Mobile \rightarrow Mobile}, m8)],\\ & sv2_{Mobile \rightarrow Mobile} \,/\, mobile(z_{Mobile}, m6), m1 \,/\, m4, m2 \,/\, m5, m3 \,/\, m7\ ]\end{aligned}$$

are unifiers of t and r, since it holds:

$$\begin{aligned}
&\sigma(t) = \mathit{weight}(\mathit{sv}1[\mathit{mobile}(\mathit{sv}2[\mathit{mobile}(\mathbf{occ}(m1), m2)], m3)])\\
&\qquad[\ \mathit{sv}1_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}3_{\mathit{Mobile} \rightarrow \mathit{Mobile}},\\
&\qquad\ \mathit{sv}2_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}4[\mathit{mobile}(\mathit{mobile}(z_{\mathit{Mobile}}, m6), m7)],\\
&\qquad\ m1 / m4, m2 / m5, m3 / m8\ ] =\\
&\ \ \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])
\end{aligned}$$

and

$$\begin{aligned}
&\sigma(r) = \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])\\
&\ \ [\ \mathit{sv}1_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}3_{\mathit{Mobile} \rightarrow \mathit{Mobile}},\\
&\qquad \mathit{sv}2_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}4[\mathit{mobile}(\mathit{mobile}(z_{\mathit{Mobile}}, m6), m7)],\\
&\qquad m1 / m4, m2 / m5, m3 / m8\ ] =\\
&\ \ \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])
\end{aligned}$$

and

$$\begin{aligned}
&\sigma'(t) = \mathit{weight}(\mathit{sv}1[\mathit{mobile}(\mathit{sv}2[\mathit{mobile}(\mathbf{occ}(m1), m2)], m3])\\
&\ \ [\ \mathit{sv}1_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}3[\mathit{mobile}(\mathit{sv}4_{\mathit{Mobile} \rightarrow \mathit{Mobile}}, m8)],\\
&\qquad \mathit{sv}2_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{mobile}(z_{\mathit{Mobile}}, m6), m1 / m4, m2 / m5, m3 / m7\ ] =\\
&\ \ \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])
\end{aligned}$$

and

$$\begin{aligned}
&\sigma'(r) = \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])\\
&\ \ [\ \mathit{sv}1_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{sv}3[\mathit{mobile}(\mathit{sv}4_{\mathit{Mobile} \rightarrow \mathit{Mobile}}, m8)],\\
&\qquad \mathit{sv}2_{\mathit{Mobile} \rightarrow \mathit{Mobile}} / \mathit{mobile}(z_{\mathit{Mobile}}, m7), m1 / m4, m2 / m5, m3 / m7\ ] =\\
&\ \ \mathit{weight}(\mathit{sv}3[\mathit{mobile}(\mathit{sv}4[\mathit{mobile}(\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m4), m5), m6), m7)], m8)])
\end{aligned}$$

But neither $\sigma \leq \sigma'$, nor $\sigma' \leq \sigma$ holds. Therefore there is no most general unifier for a set of attribute terms. Nevertheless we will see that a complete set of minimal unifiers exists!

Definition 5.1.2 (set of unifiers)

A set *unifSet* $\subseteq$ *Subst* is a *set of unifiers* for two attribute terms t and r iff for all $\sigma \in \mathit{unifSet}$ holds: $\sigma(t) \equiv \sigma(r)$.

A unifier set *unifSet* is more general than a unifier set *unifSet'* (written: $\mathit{unifSet} \leq \mathit{unifSet}'$), iff for all $\sigma' \in \mathit{unifSet}'$ exists a $\sigma \in \mathit{unifSet}$ such that $\sigma \leq \sigma'$.

By $\mathit{unifSet}(t, r)$ the set of all unifiers for t and r is denoted. ♦

Definition 5.1.3 (equivalence, variants of sets of unifiers)

Two unifier sets *unifSet* and *unifSet'* are *equivalent* (denoted by: $\mathit{unifSet} \approx \mathit{unifSet}'$), iff $\mathit{unifSet} \leq \mathit{unifSet}'$ and $\mathit{unifSet}' \leq \mathit{unifSet}$ holds.

Two unifier sets *unifSet* and *unifSet'* are *variants* (denoted by: $\mathit{unifSet} \sim \mathit{unifSet}'$), iff for all $\sigma \in \mathit{unifSet}$ there exists a $\sigma' \in \mathit{unifSet}'$ and a bijective substitution $\tau \in \mathit{Subst}$ with $\sigma = \sigma'\tau$ and vice versa.

Definition 5.1.4 (complete set of minimal unifiers)

The complete set of minimal unifiers for two attribute terms $t, r \in AT_{\Sigma}(SV)$ (written: *mguSet*(t, r)) is the minimal set in the sense of $\leq$ on sets of unifiers, i.e.

(1) *mguSet(t, r)* $\subseteq$ *unifSet(t, r)* (correctness)
(2) for all σ $\in$*unifSet(t, r)* exists a σ' $\in$*mguSet(t, r)* such that $\sigma' \leq \sigma$ (completeness)
(3) for all σ, σ' $\in$*mguSet(t, r)* with $\sigma \leq \sigma'$ holds $\sigma \equiv \sigma'$ (minimal)

Lemma 5.1.5

It holds:

unifSet ≈ *unifSet*' iff *unifSet* ~ *unifSet*'

Proof

Holds, since lemma 2.5 of [Hofbauer, Kutsche 89] at page 35 and lemma 1.13(v) at page 6 can be extended to sets of unifiers. ◆

Theorem 5.1.6 (uniqueness of the complete set of minimal unifiers)

Let *t*, *r* $\in AT_\Sigma(SV)$. If *t* and *r* are unifiable and *mguSet* and *mguSet*' are two complete sets of minimal unifiers then there exists a renaming substitution σ $\in$*Renaming* such that σ(*mguSet*) = *mguSet*', i.e. the complete set of minimal unifiers is unique up to renaming of the identifiers. ◆

Proof

Let *unifSet* and *unifSet*' be two complete sets of minimal unifiers for *t* and *r*. It holds:

unifSet ≤ *unifSet*' and *unifSet*' ≤ *unifSet*

since both sets are complete minimal sets. Implying *unifSet* ≈ *unifSet*' and with Lemma 5.1.5 holds *unifSet* ~ *unifSet*'.

Thus there exists a renaming substitution π with π(*unifSet*) = π(*unifSet*') ◆

For the existence of the complete set of minimal unifiers a constructive approach is performed.

The idea of the unification algorithm is to start with a configuration $\{ \{ t \equiv r \} \}$ with unifiers $\sigma_1, \ldots, \sigma_n$ and apply as long as possible rules until a set $\{ E_1, \ldots, E_n \}$ is obtained such that σ_i is most general unifier of E_i $(1 \leq i \leq n)$.

The unification algorithm for attribute terms is a generalization of [Martelli, Montanari 82] to handle subterm identifiers and to get a complete set of minimal unifiers.

Definition 5.1.7 (unification calculus for attributed terms)

The unification calculus $\vdash_{\text{unif}}$ for attribute terms consists of the following rules with (f: s_1, $s_2, \ldots, s_n \rightarrow s$) $\in C_{Occ} \cup F$, $t \in AT_\Sigma(SV)$, t_i, $r_i \in AT_\Sigma(SV)_{s_i}$ $(1 \leq i \leq n)$, *sv* subterm identifier, *x* usual identifier, *c* context over the notion of attribute terms:

$$(U1)\frac{\{\ldots, \{\ldots, f(t_1, \ldots, t_n) \equiv f(r_1, \ldots, r_n), \ldots\} \ldots\}}{\{\ldots, \{\ldots, t_1 \equiv r_1, \ldots, t_n \equiv r_n, \ldots\}, \ldots\}}$$

if $n \geq 0$

$$(U2)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, x \equiv t, e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1\sigma, \ldots, e_{j-1}\sigma, x \equiv t, e_{j+1}\sigma, \ldots, e_k\sigma\}, \ldots\}}$$

if x not in t and x in $e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k$ and $\sigma = [\, x / t \,]$

$$(U3)\frac{\{\ldots, \{\ldots, t \equiv x, \ldots\}, \ldots\}}{\{\ldots, \{\ldots, x \equiv t, \ldots\}, \ldots\}}$$

if t is not an identifier

$$(U4)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, x \equiv x, e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\}, \ldots\}}$$

$$(U5)\frac{\{\ldots, \{\ldots, sv[t_1, \ldots, t_n] \equiv sv[r_1, \ldots, r_n], \ldots\}, \ldots\}}{\{\ldots, \{\ldots, t_1 \equiv r_1, \ldots, t_n \equiv r_n, \ldots\}, \ldots\}}$$

$$(U6)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, sv_{s_1, s_2, \ldots, s_n \to s} \equiv c, e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1\sigma, \ldots, e_{j-1}\sigma, sv_{s_1, s_2, \ldots, s_n \to s} \equiv c, e_{j+1}\sigma, \ldots, e_k\sigma\}, \ldots\}}$$

if sv not in c and sv in $e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k$ and $\sigma = [\, sv_{s_1, s_2, \ldots, s_n \to s} / c \,]$

$$(U7)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, f(r_1, \ldots, r_m) \equiv sv[t_1, \ldots, t_n], e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_1}, \ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_u}, \ldots\}}$$

if sv not in $f(r_1, \ldots, r_n)$

$$(U8)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, sv[t_1, \ldots, t_n] \equiv f(r_1, \ldots, r_m), e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_1}, \ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_u}, \ldots\}}$$

if sv not in $f(r_1, \ldots, r_n)$

such that

$\{ E_{j1}, \ldots, E_{j_u} \} =$

$\{ \{ \; x_{01} \equiv r_1, \ldots, x_{0w_0} \equiv r_{w_0}, sv_1[t_1, \ldots, t_{v_1}] = r_{w_0+1},$

$x_{11} \equiv r_{w_0+1}, \ldots, x_{1w_1} \equiv r_{w_0+w_1+1}, sv_2[t_{v_1+1}, \ldots, t_{v_2}] \equiv r_{w_0+w_1+2}, \ldots$

$sv_p[t_{v_{p-1}+1}, \ldots, t_p] \equiv r_{w_0+\ldots+w_{n-1}+p}, x_{n1} \equiv r_{w_0+\ldots+w_{p-1}+p+1}, \ldots, x_{pw_p} \equiv r_m,$

$sv_{s_1, s_2, \ldots, s_n \to s}$[7] $\equiv$

$f\left(\overline{x_0}, sv_{1, s_1, \ldots, s_{v_1} \to s_1'}, \overline{x_1}, sv_{2 s_{v_1+1}, \ldots, s_{v_2} \to s_2'}, \overline{x_2}, \ldots, \overline{x_{p-1}}, sv_{p, s_{v_{p-1}+1}, \ldots, s_p \to s_p'}, \overline{x_p}\right) \} |$

x_i and sv_j are new identifiers and subterm identifiers, respectively ($0 \le i \le n$, $1 \le j \le n$), $sv_1[t_1, \ldots, t_{v_1}]$ and r_{w_0+1}, $sv_2[t_{v_1+1}, \ldots, t_{v_2}]$ and $r_{w_0+w_1+2}, \ldots,$ $sv_p[t_{v_{p-1}+1}, \ldots, t_p]$ and $r_{w_0+\ldots+w_{n-1}+p}$ are unifiable, $0 \le v_1 < v_2 < \ldots < v_p \le n$, $0 \le w_i \le n \} \cup$

$\{ \{ t_1 \equiv f(r_1, \ldots, r_m), sv \equiv z \} \mid n = 1, t_1$ and $f(r_1, \ldots, r_m)$ are unifiable $\}$

$$(U9)\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, sv[t_1, \ldots, t_n] \equiv sv'[r_1, \ldots, r_m], e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_1}, \ldots, \{e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_k\} \cup E_{j_u}, \ldots\}},$$

if sv is not in $sv'[r_1, \ldots, r_m]$ and sv' is not in $sv[t_1, \ldots, t_n]$.

such that

$\{ E_{j1}, \ldots, E_{j_u} \} =$

$\{ \{ x_{01} \equiv r_1, \ldots, x_{0w_0} \equiv r_{w_0}, sv_1[t_1, \ldots, t_{v_1}] \equiv r_{w_0+1},$

[7] Note, $sv_{s_1, s_2, \ldots, s_n \to s}$ is an abbreviation for $sv_{s_1, s_2, \ldots, s_n \to s}[z_{s_1}, \ldots, z_{s_n}]$.

$x_{11} \equiv r_{w_0+1}, \ldots, x_{1w_1} \equiv r_{w_0+w_1+1}, sv_2[t_{v_1+1}, \ldots, t_{v_2}] \equiv r_{w_0+w_1+2}, \cdots$
$sv_p[t_{v_{p\text{-}1}+1}, \ldots, t_p] \equiv r_{w_0+\ldots+w_{n\text{-}1}+p}, x_{n1} \equiv r_{w_0+\ldots+w_{p\text{-}1}+p+1}, \ldots, x_{pw_p} \equiv r_m,$
$sv_{s_1, s_2, \ldots, s_n \to s} \equiv$

$$sv'\left[\overline{x_0}, sv_{1, s_1, \ldots, s_{v_1} \to s_1'}, \overline{x_1}, sv_{2 s_{v_1+1}, \ldots, s_{v_2} \to s_2'}, \overline{x_2}, \ldots, \overline{x_{p-1}}, sv_{p, s_{v_{p-1}+1}, \ldots, s_p \to s_p'}, \overline{x_p}\right] \; \} \; |$$

x_i and sv_j are new identifiers and subterm identifiers, respectively ($0 \le i \le n$, $1 \le j \le n$), $sv_1[t_1, \ldots, t_{v_1}]$ and r_{w_0+1}, $sv_2[t_{v_1+1}, \ldots, t_{v_2}]$ and $r_{w_0+w_1+2}, \ldots,$ $sv_p[t_{v_{p\text{-}1}+1}, \ldots, t_p]$ and $r_{w_0+\ldots+w_{n\text{-}1}+p}$ are unifiable, $0 \le v_1 < v_2 < \ldots < v_p \le n$, $0 \le w_i \le n$ } ∪

$\{ \{ t_1 \equiv sv'[r_1, \ldots, r_m], sv'_{s \to s} \equiv z_s \} \mid n = 1, t_1 \equiv sv'[r_1, \ldots, r_m]$ are unifiable } ∪

$\{ \{ x_{01} \equiv t_1, \ldots, x_{0w_0} \equiv t_{w_0}, sv_1[r_1, \ldots, r_{v_1}] \equiv t_{w_0+1},$
$x_{11} \equiv t_{w_0+1}, \ldots, x_{1w_1} \equiv t_{w_0+w_1+1}, sv_2[r_{v_1+1}, \ldots, r_{v_2}] \equiv t_{w_0+w_1+2}, \cdots$
$sv_p[r_{v_{p\text{-}1}+1}, \ldots, r_p] \equiv t_{w_0+\ldots+w_{n\text{-}1}+p}, x_{p1} \equiv t_{w_0+\ldots+w_{p\text{-}1}+p+1}, \ldots, x_{pw_p} \equiv t_n,$
$sv'_{s_1, s_2, \ldots, s_n \to s} \equiv$

$$sv\left[\overline{x_0}, sv_{1, s_1, \ldots, s_{v_1} \to s_1'}, \overline{x_1}, sv_{2 s_{v_1+1}, \ldots, s_{v_2} \to s_2'}, \overline{x_2}, \ldots, \overline{x_{p-1}}, sv_{p, s_{v_{p-1}+1}, \ldots, s_p \to s_p'}, \overline{x_p}\right] \} \; |$$

x_i and sv_j are new identifiers and subterm identifiers, respectively ($0 \le i \le n$, $1 \le j \le n$), $sv_1[r_1, \ldots, r_{v_1}]$ and t_{w_0+1}, $sv_2[r_{v_1+1}, \ldots, r_{v_2}]$ and $t_{w_0+w_1+2}, \ldots,$ $sv_p[r_{v_{p\text{-}1}+1}, \ldots, r_p]$ and $t_{w_0+\ldots+w_{n\text{-}1}+p}$
are unifiable, $0 \le v_1 < v_2 < \ldots < v_p \le m$, $0 \le w_i \le m$ } ∪

$\{ \{ r_1 \equiv sv[t_1, \ldots, t_n], sv \equiv z \} \mid m = 1, r_1 \equiv sv'[t_1, \ldots, t_n]$ are unifiable }

Remarks:

Note, that (U1)-(U4) is the usual unification, (U5) is the extension of (U1) to subterm identifiers and (U6) the extension of (U2) to subterm identifiers. (U7)-(U9) are rules for subterm identifiers. (U8) is the reverse of (U7) therefore a rule

$$(U7')\frac{\{\ldots, \{e_1, \ldots, e_{j-1}, f(r_1, \ldots, r_m) \equiv sv[t_1, \ldots, t_n], e_{j+1}, \ldots, e_k\}, \ldots\}}{\{\ldots, \{e_1, \ldots, e_{j-1}, sv[t_1, \ldots, t_n] \equiv f(r_1, \ldots, r_m), e_{j+1}, \ldots, e_k\}, \ldots\}}$$

is enough, but then the proof of termination is more complicated.

The idea of (U7)-(U9) is the same. All possible splittings of the subterm identifier dependending on the right hand side of the equation have to be constructed. All these cases have to be considered in the rule. But the number of splitting is restricted, because only sort-correct terms are assumed.

Definition 5.1.8 (unifiablity of a set of sets of equations)

A set of sets of equations $SE = \{ \{ t_{11} \equiv r_{11}, \ldots, t_{1n_1} \equiv r_{1n_1} \}, \ldots, \{ t_{k1} \equiv r_{k1}, \ldots, t_{kn_k} \equiv r_{kn_k} \} \}$ is unifiable with a set of unifiers $unifset = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ iff

(i) $\forall$ *se* $\in$ *SE*. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se* and

(ii) $\forall\, \sigma \in$ *unifset*. $\exists$ *se* $\in$ *SE*. σ is unifier of *se*. ♦

Lemma 5.1.9 (unification invariance)

Let SE_i be a set of set of equations and SE_{i+1} the set of set of equations obtained by applying a rule $(U1)$-$(U9)$ to SE_i. It holds:

(1) SE_i is unifiable with a set of unifiers *unifset*, if SE_{i+1} is unifiable with *unifset*.

(2) If *unifset* is a complete set of minimal unifiers of SE_i, then *unifset* is a complete set of minimal unifiers of SE_{i+1}. ♦

Proof of (1)

It holds:

$\forall$ *se* $\in SE_i$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*

implies

$\forall$ *se* $\in SE_i \cap SE_{i+1}$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*

Thus it is sufficient to show:

$\forall$ *se* $\in SE_i$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*

implies

$\forall$ *se* $\in SE_{i+1} \setminus SE_i$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*

$(U1)$ (i)

It holds:

$SE_{i+1} = (SE_i \setminus \{e_1,\dots, e_{j-1}, f(t_1, t_2,\dots, t_n) \equiv f(r_1, r_2,\dots, r_n), e_{j+1},\dots, e_k\})$
$\cup \{e_1,\dots, e_{j-1}, t_1 \equiv r_1,\dots, t_n \equiv r_n, e_{j+1},\dots, e_k\}$

Since $\forall$ *se* $\in SE_i$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*, especially for
$\{e_1,\dots, e_{j-1}, f(t_1, t_2,\dots, t_n) \equiv f(r_1, r_2,\dots, r_n), e_{j+1},\dots, e_k\}$
exists a substitution in *unifset*. Let $\sigma \in$ *unifset* be such a substitution.
σ is also a unifier of the set
$\{e_1,\dots, e_{j-1}, t_1 \equiv r_1,\dots, t_n \equiv r_n, e_{j+1},\dots, e_k\}$
because the application of the substitution is defined inductively.

(ii)

Let $\sigma \in$ *unifset* be an arbitrary substitution. There exists an *se* $\in SE_i$ such that σ is unifier of *se*. If *se* $\neq \{e_1,\dots, e_{j-1}, f(t_1, t_2,\dots, t_n) \equiv f(r_1, r_2,\dots, r_n), e_{j+1},\dots, e_k\}$ then there exists also an equation in SE_{i+1} such that σ is unifier of the equation.

Otherwise let us assume that
σ is unifier of $\{e_1,\dots, e_{j-1}, f(t_1, t_2,\dots, t_n) \equiv f(r_1, r_2,\dots, r_n), e_{j+1},\dots, e_k\}$

Again since the application of the substitution is inductively defined, σ is also a unifier of t_i and r_i $(0 \leq i \leq n)$ and therefore there exists also a *se* $\in SE_{i+1}$ such that σ is unifier of *se*, namely
se $= \{e_1,\dots, e_{j-1}, t_1 \equiv r_1,\dots, t_n \equiv r_n, e_{j+1},\dots, e_k\}$.

$(U2)$ (i)

It holds:

$SE_{i+1} = (SE_i \setminus \{ e_1,\dots, e_{j-1}, x \equiv t, e_{j+1},\dots, e_k\}) \cup \{ e_1\sigma,\dots, e_{j-1}\sigma, x \equiv t, e_{j+1}\sigma,\dots, e_k\,\sigma \}$

Since $\forall$ *se* $\in SE_i$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*, especially for
$\{ e_1,\dots, e_{j-1}, x \equiv t, e_{j+1},\dots, e_k \}$

there exists a substitution in *unifset*. Let $\sigma \in unifset$ be such a substitution. Furthermore, $\tau = [\, x \,/\, t \,]$ is a unifier for $x \equiv t$. Thus σ can be written as $\sigma = \tau\pi$ for some substitution π. Because x is not in t, it holds: $\tau\tau = \tau$

Therefore $\sigma = \tau\pi = \tau\tau\pi = \tau\sigma$ is valid leading to the fact that σ is unifier of

$\{\, e_1, \dots, e_{j-1}, x \equiv t, e_{j+1}, \dots, e_k \,\}$

iff

$\{\, e_1\sigma, \dots, e_{j-1}\sigma, x \equiv t, e_{j+1}\sigma, \dots, e_k\,\sigma \,\}$

is unifiable with σ. Therefore σ is also unifier of

$\{\, e_1\sigma, \dots, e_{j-1}\sigma, x \equiv t, e_{j+1}\sigma, \dots, e_k\,\sigma \,\}$

(ii)

Let $\sigma \in unifset$ be an arbitrary substitution, then there exists an $se \in SE_i$ such that σ is a unifier of se. If $se \neq \{\, e_1, \dots, e_{j-1}, x \equiv t, e_{j+1}, \dots, e_k \,\}$ then there exists an equation in SE_{i+1} such that σ is unifier of this equation.

Otherwise let us assume that

σ is unifier of $\{\, e_1, \dots, e_{j-1}, x \equiv t, e_{j+1}, \dots, e_k \,\}$

Since $\{\, e_1, \dots, e_{j-1}, x \equiv t, e_{j+1}, \dots, e_k \,\}\sigma = \{\, e_1\sigma, \dots, e_{j-1}\sigma, x\sigma \equiv t\sigma, e_{j+1}\sigma, \dots, e_k\sigma \,\}$ is valid σ is also unifier of $se = \{\, e_1\sigma, \dots, e_{j-1}\sigma, x \equiv t, e_{j+1}\sigma, \dots, e_k\,\sigma \,\}$ and therefore there exists a $se \in SE_{i+1}$ such that σ is unifier of se.

($U3$) and ($U4$) satisfy trivially the unification invariance.

($U5$) can be proved analogous to ($U1$).

($U6$) can be proved analogous to ($U2$).

($U7$) (i)

It holds:

$SE_{i+1} = (SE_i \setminus \{\, e_1, \dots, e_{j-1}, f(r_1, \dots, r_m) \equiv sv[t_1, \dots, t_n], e_{j+1}, \dots, e_k \,\})$
$\quad \cup\, (\cup_{1 \le i \le u} (\{\, e_1, \dots, e_{j-1}, e_{j+1}, \dots, e_k \,\} \cup E_{j_i})$

Since $\forall\, se \in SE_i.\ \exists\, \sigma \in unifset.\ \sigma$ is unifier of se, especially for

$\{\, e_1, \dots, e_{j-1}, f(r_1, \dots, r_m) \equiv sv[t_1, \dots, t_n], e_{j+1}, \dots, e_k \,\}$

there exists a substitution in *unifset*. Let $\sigma \in unifset$ be such a substitution.

Since attributed terms are defined inductively, σ is a unifier of $f(r_1, \dots, r_m)$ and $sv[t_1, \dots, t_n]$ if either (notations as in Definition 5.1.7)

(a) $f(r_1, \dots, r_m)$ and t_1 are unifiable with σ, i.e. sv is trivial,

or

(b) $sv[t_1, \dots, t_n]$ has the form

$f(\overline{x_0}, sv_1[t_1, \dots, t_{v_1}], \overline{x_1}, sv_2[t_{v_1+1}, \dots, t_{v_2}], \overline{x_2}, \dots, \overline{x_{p-1}}, sv_p[t_{v_{p-1}+1}, \dots, t_p], \overline{x_p})$

(a)

Let $\tau = [\, sv \,/\, z \,]$. σ can be written as $\sigma = \tau\,\pi$ for some substitution π. It holds:

$\sigma = \tau\,\pi = \tau\,\tau\,\pi = \tau\,\sigma$ (since $\tau = \tau\,\tau$)

Therefore σ is also unifier of $\{\, e_1, \dots, e_{j-1}, t_1 \equiv f(r_1, \dots, r_m), sv \equiv z, e_{j+1}, \dots, e_k \,\}$

(b)

If $sv[t_1, \dots, t_n]$ has the form

$f(\overline{x_0}, sv_1[t_1, \dots, t_{v_1}], \overline{x_1}, sv_2[t_{v_1+1}, \dots, t_{v_2}], \overline{x_2}, \dots, \overline{x_{p-1}}, sv_p[t_{v_{p-1}+1}, \dots, t_p], \overline{x_p})$

then $f(r_1, \dots, r_m)$

and $f(\overline{x_0}, sv_1[t_1, \dots, t_{v_1}], \overline{x_1}, sv_2[t_{v_1+1}, \dots, t_{v_2}], \overline{x_2}, \dots, \overline{x_{p-1}}, sv_p[t_{v_{p-1}+1}, \dots, t_p], \overline{x_p})$ are unifiable if

$\sigma(x_{01}) \equiv \sigma(r_1),\ldots, \sigma(x_{0w_0}) \equiv \sigma(r_{w_0}), \sigma(sv_1[t_1,\ldots, t_{v_1}]) \equiv \sigma(r_{w_0+1}),$
$\sigma(x_{11}) \equiv \sigma(r_{w_0+1}),\ldots, \sigma(x_{1w_1}) \equiv \sigma(r_{w_0+w_1+1}), \sigma(sv_2[t_{v_1+1},\ldots, t_{v_2}]) \equiv \sigma(r_{w_0+w_1+2}),\ldots$
$\sigma(sv_p[t_{v_{p-1}+1},\ldots, t_p]) \equiv \sigma(r_{w_0+\ldots+w_{n-1}+p}), \sigma(x_{n1}) \equiv \sigma(r_{w_0+\ldots+w_{p-1}+p+1}),\ldots,$
$\sigma(x_{1w_p}) \equiv \sigma(r_m) \}$
Let
$\tau = [\, sv /$

$$f\left(\overline{x_0}, sv_{1,s_1,\ldots,s_{v_1}\to s_1'}, \overline{x_1}, sv_{2s_{v_1+1},\ldots,s_{v_2}\to s_2'}, \overline{x_2}, \ldots, \overline{x_{p-1}}, sv_{p,s_{v_{p-1}+1},\ldots,s_p\to s_p'}, \overline{x_p}\right)]$$

then σ can be written as $\sigma = \tau\,\pi$ for some substitution π. Therefore σ is also unifier of
$\{\, x_{01} \equiv r_1,\ldots, x_{0w_0} \equiv r_{w_0}, sv_1[t_1,\ldots, t_{v_1}] \equiv r_{w_0+1},$
$x_{11} \equiv r_{w_0+1},\ldots, x_{1w_1} \equiv r_{w_0+w_1+1}, sv_2[t_{v_1+1},\ldots, t_{v_2}] \equiv r_{w_0+w_1+2},\ldots$
$sv_p[t_{v_{p-1}+1},\ldots, t_p] \equiv r_{w_0+\ldots+w_{n-1}+p}, x_{n1} \equiv r_{w_0+\ldots+w_{p-1}+p+1},\ldots, x_{1w_p} \equiv r_m,$
$sv[z_1,\ldots, z_n] \equiv$

$$f\left(\overline{x_0}, sv_{1,s_1,\ldots,s_{v_1}\to s_1'}, \overline{x_1}, sv_{2s_{v_1+1},\ldots,s_{v_2}\to s_2'}, \overline{x_2}, \ldots, \overline{x_{p-1}}, sv_{p,s_{v_{p-1}+1},\ldots,s_p\to s_p'}, \overline{x_p}\right) \}$$

Thus $\forall$ *se* $\in SE_{i+1}$. $\exists\, \sigma \in$ *unifset*. σ is unifier of *se*.
(ii) analogous to (i).

(*U*8) analogous to (*U*7).

(*U*9) analogous to (*U*7).

Proof of (2)

It was shown in (1) that every set of unifiers of SE_i is a set of unifiers of SE_{i+1}. It remains to show that the set is complete and minimal. Considering the rules they do not change both properties. ◆

Definition 5.1.10 (unification algorithm)

The unification algorithm for the unification of attribute terms is defined by the following three steps:

(1) The start configuration is $\{ \{ t \equiv r \} \}$.
(2) Apply rule (*U*1)-(*U*9) until no more reductions are possible.
(3) Let *E* be the solution of step (2). ◆

Lemma 5.1.11 (termination)

The unification algorithm terminates. ◆

Proof

Take the following termination function mapping a set of sets of equations *SE* to a triple

(k, l, m)

such that

$$k = \big|\, var(SE) \setminus (\{ x \mid x \text{ in } SE \text{ only once in an equation of the form } x \equiv t \} \cup \{ sv \mid sv \text{ in } SE \text{ only once in an equation of the form } sv_{s_1, s_2,\ldots,s_n\to s} \equiv t \}) \,\big|$$

l = Sum of the sizes of all equations

$m = \Big|$ $\{ x \mid x$ in SE only once in an equation of the form $x \equiv t \} \cup$

$\{ sv \mid sv$ in SE only once in an equation of the form $sv_{s_1, s_2,\ldots, s_n \to s} \equiv t \}$ $\Big|$

with the lexicographical ordering.

The size of an equation is defined as the size of the attribute terms on the left- and right-hand side of an equation being defined by

$size(f(t_1,\ldots, t_n)) = (size(t_1) + \ldots + size(t_n) + 1) * height(f(t_1,\ldots, t_n))$
$size(sv[t_1,\ldots, t_n]) = (size(t_1) + \ldots + size(t_n) + 1) * height(f(t_1,\ldots, t_n)) * combinations$[8] $+ 1$
$height(f) = 1$
$height(f(t_1,\ldots, t_n)) = max \{ height(t_1),\ldots, height(t_n) \} + 1$

It holds:

$(U1)$ k constant, l smaller
$(U2)$, $(U3)$, $(U4)$ k smaller
$(U5)$ k constant, l smaller
$(U6)$ k smaller
$(U7)$, $(U8)$ k constant, l smaller
$(U9)$ k constant, l smaller ◆

Theorem 5.1.12 (existence, calculation of the complete set of minimal unifiers)

Let t and r be two attribute terms. Let SE_0 be the start configuration $\{ \{ t \equiv r \} \}$ and SE_n be the result of the unification algorithm. It holds:

(1) If each $se \in SE_n$ has the form

$\{ x_1 \equiv t_1,\ldots, x_n \equiv t_n, sv_{1, s_1, s_2,\ldots, s_n \to s_1'} \equiv c_1,\ldots, sv_{k, s_1, s_2,\ldots, s_n \to s_k'} \equiv c_k \}$

and the x_i's and sv_j's are all pairwise distinct and all x_i and sv_j do not appear in t_l and c_m (then se is called *completely solved*) then t and r are unifiable with the complete set of minimal unifiers (*associated substitution*):

$[x_1 / t_1,\ldots, x_n / t_n, sv_{1, s_1, s_2,\ldots, s_n \to s_1'} / c_1,\ldots, sv_{k, s_1, s_2,\ldots, s_n \to s_k'} / c_k]$

(2) Otherwise t and r are not unifiable. ◆

Proof

Since the unification algorithm terminates (Lemma 5.1.11) there exists an end configuration SE_n for every start configuration SE_0.

proof of (1)

Let SE_n be completely solved. SE_n is unifiable with the associated substitution of SE_n, since for all $se \in SE_n$ holds:

se has the form

$\{ x_1 \equiv t_1,\ldots, x_n \equiv t_n, sv_{1, s_1, s_2,\ldots, s_n \to s_1'} \equiv c_1,\ldots, sv_{k, s_1, s_2,\ldots, s_n \to s_k'} \equiv c_k \}$

[8] By *combinations* the number of possible splittings of a subterm identifier is meant.

and

$$\tau = \{ x_1 / t_1,\ldots, x_n / t_n, sv_{1,s_1, s_2,\ldots, s_n \rightarrow s_1'} / c_1,\ldots, sv_{k, s_1, s_2,\ldots, s_n \rightarrow s_k'} / c_k \}$$

is unifier of *se*, because $x_i\tau \equiv t_i\,\tau$ and $sv_i\tau \equiv c_i\,\tau$.

Let π be an arbitrary unifier of *se*. We show $\tau \leq \pi$ by showing $\tau\,\pi = \pi$. Therefore τ is *minimal unifier* for *se* in SE_n and of SE_0 because of the invariance lemma and thus of *t* and *r*. Moreover, SE_n is a *complete* set since *se* and π are arbitrary.

To show

- $x\,\tau\,\pi = x\,\pi$ holds for all identifiers *x* and
- $sv\,\tau\pi \equiv sv\,\pi$ holds for all subterm identifiers *sv*.

proof

$x \notin$dom(τ): $x\,\tau\,\pi \equiv x\,\pi$, since $x\,\tau \equiv x$

$x \in$dom(τ): i.e. $x \equiv x_i$ for some x_i. It holds: $x\,\tau\,\pi \equiv x_i\,\tau\,\pi \equiv t_i\,\pi$
Moreover, since π is unifier of x_i and t_i it holds: $t_i\,\pi \equiv x_i\,\pi \equiv x\,\pi$
and therefore: $x\,\tau\,\pi \equiv x_i\,\tau\,\pi \equiv t_i\,\pi \equiv x_i\,\pi \equiv x\,\pi$

$sv \notin$dom(τ): $sv\,\tau\pi \equiv sv\,\pi$, since $sv\,\tau \equiv sv$

$sv \in$dom(τ): i.e. $sv \equiv sv_i$ for some sv_i.
It holds: $sv\,\tau\pi \equiv sv_i\,\tau\pi \equiv c_i\,\pi$
Moreover, since π is unifier of sv_i and c_i it holds:
$c_i\,\pi \equiv sv_i\,\pi \equiv sv\,\pi$
and therefore: $sv\,\tau\,\pi \equiv sv\,\pi$

proof of (2)

Let SE_n be not completely solved and no rule be applicable to SE_n, then there exists an equation $t \equiv r$ in SE_n of the form

either $f(t_1,\ldots, t_n) \equiv g(r_1,\ldots, r_m)$ and $f \not\equiv g$
or $x \equiv t$ and *x* in *t*
or $sv[t_1,\ldots, t_n] \equiv r$ and *sv* in *r*
or $f \equiv sv[r_1,\ldots, r_m]$ with $m > 1$
or the cases of (*U*7)-(*U*9).

In all these cases the terms are not unifiable. ♦

The result of these lemmas can be summarized in the following theorem:

Theorem 5.1.13 (soundness and completeness)

The unification algorithm is sound and complete. ♦

Example 5.1.14

Let us consider the terms

t = *mobile*(**occ**(*cube*(*n*)), *mobile*(*cube*(2), *cube*(3))) and

r = *sv*1[*mobile*(**occ**(*m*1), *m*2)]

The derivation looks like

$$(U7)\frac{\{\{mobile(occ(cube(n)), mobile(cube(2), cube(3))) \equiv sv1[mobile(occ(m1), m2)]\}\}}{\{\{\ sv1 \equiv z,}$$

$$(U1)\frac{mobile(occ(cube(n)), mobile(cube(2), cube(3))) \equiv mobile(occ(m1), m2)\}\}}{\{\{\ sv1 \equiv z,}$$

$$(U3)\frac{occ(cube(n)) \equiv occ(m1), mobile(cube(2), cube(3)) \equiv m2\}\}}{\{\{\ sv1 \equiv z,}$$

$$(U1)\frac{occ(cube(n)) \equiv occ(m1), m2 \equiv mobile(cube(2), cube(3))\}\}}{\{\{\ sv1 \equiv z,}$$

$$(U3)\frac{cube(n) \equiv m1, m2 \equiv mobile(cube(2), cube(3))\}\}}{\{\{\ sv1 \equiv z,}$$

$$m1 \equiv cube(n), m2 \equiv mobile(cube(2), cube(3))\}\}$$

and the obtained substitution is

$[\ sv1 \,/\, z, m1 \,/\, cube(n), m2 \,/\, mobile(cube(2), cube(3))\ ]$

For the terms

$t = sv[cube(n), cube(2), cube(3)]$

and

$r = mobile(m1, m2)$

the derivation is

$$(U8)\frac{\{\{sv[cube(n), cube(2), cube(3)] \equiv mobile(m1, m2)\}\}}{\{\{sv1[cube(n), cube(2), cube(3)] \equiv m1, m3 \equiv m2,}$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile, Mobile, Mobile \to Mobile}, m3)\},$$

$$\{m \equiv m1, sv1[cube(n), cube(2), cube(3)] \equiv m2,$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(m3, sv1_{Mobile, Mobile, Mobile \to Mobile})\},$$

$$\{sv1[cube(n), cube(2)] \equiv m1, sv2[cube(3)] \equiv m2,$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile, Mobile \to Mobile}, sv2_{Mobile \to Mobile})\},$$

$$\{sv1[cube(n)] \equiv m1, sv2[cube(2), cube(3)] \equiv m2,$$

$$(U3^{*})\frac{sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile \to Mobile}, sv2_{Mobile, Mobile \to Mobile})\}\}}{\{\{m1 \equiv sv1[cube(n), cube(2), cube(3)], m3 \equiv m2,}$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile, Mobile, Mobile \to Mobile}, m3)\},$$

$$\{m3 \equiv m1, m2 \equiv sv1[cube(n), cube(2), cube(3)],$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(m3, sv1_{Mobile, Mobile, Mobile \to Mobile})\},$$

$$\{m1 \equiv sv1[cube(n), cube(2)], m2 \equiv sv2[cube(3)],$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile, Mobile \to Mobile}, sv2_{Mobile \to Mobile})\},$$

$$\{m1 \equiv sv1[cube(n)], m2 \equiv sv2[cube(2), cube(3)],$$

$$sv_{Mobile, Mobile, Mobile \to Mobile} \equiv mobile(sv1_{Mobile \to Mobile}, sv2_{Mobile, Mobile \to Mobile})\}\}$$

and the obtained substitutions are

$[\ m1 \,/\, sv1[cube(n), cube(2), cube(3)], m3 \,/\, m2, sv \,/\, mobile(sv1, m3)\ ],$
$[\ m3 \,/\, m1, m2 \,/\, sv1[cube(n), cube(2), cube(3)], sv \,/\, mobile(m3, sv1)\ ],$
$[\ m1 \,/\, sv1[cube(n), cube(2)], m2 \,/\, sv2[cube(3)], sv \,/\, mobile(sv1, sv2)\ ],$
$[\ m1 \,/\, sv1[cube(n)], m2 \,/\, sv2[cube(2), cube(3)], sv \,/\, mobile(sv1, sv2)\ ].$ ♦

5.2 Universally Quantified Formulae

In this subsection we present an extension of the equational calculus for proving universally quantified formulae and define an induction principle for occurrence terms afterwards. An induction ordering on occurrence terms is defined as a generalization. Finally, we formulate a semi-algorithm for constructing complete sets of occurrence terms and discuss heuristics for the proof principle.

5.2.1 Attributed Equational Calculus

The attributed equational calculus is an extension of the equational calculus with infinite induction (cf. e.g. [Wirsing 90]) to handle subterm identifiers.

Definition 5.2.1.1 (attributed equational calculus)

The attributed equational calculus $\vdash_{\text{aeq}}$ consists of the following derivation rules:

Let *t*, *r*, *u* $\in AT_\Sigma(SV)_s$ be attribute terms, $(f\colon s_1, s_2, \ldots, s_n \to s) \in C \cup F$ and $t_i \in AT_\Sigma(SV)_{s_i}$ $(1 \le i \le n)$, *sv* be a subterm identifier, *c* be a context and *Ax* be a set of axioms.

$$(ax)\frac{}{t = r} \quad \text{if } t = r \in Ax$$

$$(refl)\frac{}{t = t}$$

$$(sym)\frac{t = r}{r = t}$$

$$(trans)\frac{t = r \quad r = u}{t = u}$$

$$(fun)\frac{t_1 = r_1, t_2 = r_2, \ldots, t_n = r_n}{f(t_1, t_2, \ldots, t_n) = f(r_1, r_2, \ldots, r_n)}$$

$$(subst)\frac{t = r}{t[sv_{s_1, s_2, \ldots, s_n \to s}/c] = r[sv_{s_1, s_2, \ldots, s_n \to s}/c]}$$

$$(II)\frac{(t = r)[sv/c] \text{ for all } c \in ST_{(S, C, \varnothing)}(SV')_s}{\forall sv_{s_1, s_2, \ldots, s_n \to s} \cdot (t = r)}$$

with $SV' = (SV_{s_1, s_2, \ldots, s_n \to s})_{s_1, s_2, \ldots, s_n, s \in S^*, S \setminus \{s\}}$,
$t\,[sv / c], r\,[sv / c] \in AT_\Sigma(SV)$ ♦

In the following we show the soundness and completeness of the attributed equational calculus.
Firstly, the soundness of the attributed equational calculus is shown:

Theorem 5.2.1.2 (soundness of the attributed equational calculus)

Let *ASpec* = $<\Sigma, F_{Attr}, Ax>$ be an attributed algebraic specification associated with the axioms *Ax*. The attributed equational calculus $\vdash_{\text{aeq}}$ is sound wrt. *Mod(ASpec)*, i.e.

$Ax \vdash_{\text{aeq}} t = r$ implies $Mod(ASpec) \models_{\text{attr}} t = r$ ♦

Proof

Using fact 5.2.1 of [Ehrig, Mahr 85] it is sufficient to show that the rules of the attributed equational calculus are correct.

Let A be an arbitrary model for the axioms Ax and v an arbitrary valuation for A.

(ax) is valid since $I^A_v\,[t] = I^A_v\,[r]$ and therefore $Mod(ASpec) \models_{\text{attr}} t = r$, because A is a model of $ASpec$ satisfying all axioms.

($refl$) is valid since $I^A_v\,[t] = I^A_v\,[t]$ and therefore $Mod(ASpec) \models_{\text{attr}} t = t$

(sym) is valid since $I^A_v\,[t] = I^A_v\,[r]$ implies $I^A_v\,[r] = I^A_v\,[t]$ and therefore $Mod(ASpec) \models_{\text{attr}} r = t$

($trans$) $A, v \models_{\text{attr}} t = r$ and $A, v \models_{\text{attr}} r = u$, i.e. $I^A_v\,[t] = I^A_v\,[r]$ and $I^A_v\,[r] = I^A_v\,[u]$ is valid and therefore $I^A_v\,[t] = I^A_v\,[u]$ implying $Mod(ASpec) \models_{\text{attr}} t = u$.

($subst$) $A, v \models_{\text{attr}} t = r$, i.e. $I^A_v\,[t] = I^A_v\,[r]$ is valid for all valuations v. Especially for a valuation v' with $v'(sv') = v(sv')$ with $sv' \in SV \setminus \{\ sv\ \}$ and $v'(sv) = I^A_v\,[c]$ holds $A, v' \models_{\text{attr}}\ t = r$, i.e. $I^A_{v'}\,[t] = I^A_{v'}\,[r]$. Since the substitution and the interpretation are analogously defined, it follows $I^A_v\,[t\ [sv\ /\ c]] = I^A_{v'}\,[t] = I^A_{v'}\,[r] = I^A_v\,[r\ [sv\ /\ c]]$ (complete proof by induction on the attribute term notion) and therefore $A, v \models_{\text{attr}} t\ [sv\ /\ c] = r\ [sv\ /\ c]$ holds for all valuations v.

(fun) $A, v \models_{\text{attr}} t_1 = r_1, A, v \models_{\text{attr}} t_2 = r_2, \ldots, A, v \models_{\text{attr}} t_n = r_n$, i.e. $I^A_v\,[t_1] = I^A_v\,[r_1], I^A_v\,[t_2] = I^A_v\,[r_2], \ldots, I^A_v\,[t_n] = I^A_v\,[r_n]$ and therefore $I^A_v\,[f(t_1, t_2, \ldots, t_n)] = f^A(I^A_v\,[t_1], I^A_v\,[t_2], \ldots, I^A_v\,[t_n]) = f^A(I^A_v\,[r_1], I^A_v\,[r_2], \ldots, I^A_v\,[r_n]) = I^A_v\,[f(r_1, r_2, \ldots, r_n)]$ and thus $A, v \models_{\text{attr}} f(t_1, t_2, \ldots, t_n) = f(r_1, r_2, \ldots, r_n)$, i.e. $Mod(ASpec) \models_{\text{attr}} f(t_1, t_2, \ldots, t_n) = f(r_1, r_2, \ldots, r_n)$

(II) Let $A, v \models_{\text{attr}} t\ [sv\ /\ c] = r\ [sv\ /\ c]$ for all $c \in ST_{(S,\, C,\, \varnothing)}(SV')_s$, i.e. $I^A_v\,[t\ [sv\ /\ c]] = I^A_v\,[r\ [sv\ /\ c]]$ for all $c \in ST_{(S,\, C,\, \varnothing)}(SV')_s$, i.e. for all valuations v' with $v'(sv') = v(sv')$ with $sv' \in SV \setminus \{\ sv\ \}$ and $v'(sv') = I^A_v\,[c]$ holds: $I^A_{v'}\,[t] = I^A_{v'}\,[r]$
Thus $A, v' \models_{\text{attr}} \forall\ sv_{s_1, s_2, \ldots, s_n \rightarrow s}.\ t = r$ is valid because the considered algebras are reachable and bacause of the definition of v'.
$Mod(ASpec) \models_{\text{attr}} \forall\ sv_{s_1, s_2, \ldots, s_n \rightarrow s}.\ t = r$ ◆

Moreover, the attributed equational calculus is complete.

Theorem 5.2.1.3 (completeness of the attributed equational calculus)

Let $ASpec = \langle\Sigma, F_{Attr}, Ax\rangle$ be an attributed algebraic specification associated with the axioms Ax. The attributed equational calculus $\vdash_{\text{aeq}}$ is complete wrt. $Mod(ASpec)$, i.e.

$Mod(ASpec) \models_{\text{attr}}\ t = r$ implies $Ax \vdash_{\text{aeq}} t = r$ ◆

Proof

The proof of the completeness of the attributed equational calculus is similar to the proof of the completeness of the equational calculus given e.g. in [Ehrich et al. 89] and of the equational calculus with infinite induction given in [Wirsing 90].

In order to show the completeness of the attributed equational calculus the following congruence $\sim_s$ is defined on $AT_\Sigma(SV)_s$ for all $s \in S$ with $SV = (SV_{s_1, s_2,\ldots, s_n \rightarrow s})_{s_1, s_2,\ldots, s_n, s \in S^*, S \setminus \{s\}}$:

$t \sim_s r$ iff $Ax \vdash_{\text{aeq}} t = r$ for some $t, r \in AT_\Sigma(SV)_s$

The fact, that $\sim_s$ defines a congruence relation, immediately results from the rules (*refl*), (*sym*), (*trans*) and (*fun*).

For all terms $t, r \in AT_\Sigma(SV)_s$ holds: $Mod(ASpec) \models_{\text{attr}} t = r$ implies $Ax \vdash_{\text{aeq}} t = r$

Since:

Obviously $AT_\Sigma(SV)_s / \sim_s$ satisfies the axioms, i.e. $AT_\Sigma(SV)_s / \sim_s \in Mod(ASpec)$.

Therefore $Mod(ASpec) \models_{\text{attr}} t = r$ implies $AT_\Sigma(SV)_s / \sim_s \models_{\text{attr}} t = r$ implies $Ax \vdash_{\text{aeq}} t = r$

Let $t, r \in AT_\Sigma(SV)_s$ such that $Mod(ASpec) \models_{\text{attr}} t = r$

For each instance of $t = r$ there exists a derivation with (*refl*), (*sym*), (*trans*), (*fun*) and (*subst*). Applying the rule (II) yields a proof for $t = r$ by $\vdash_{\text{aeq}}$. ♦

Thus we have shown:

Corollary 5.2.1.4 (soundness, completeness of the attributed equational calculus)

The attributed equational calculus is sound and complete. ♦

5.2.2 Attributed Term Induction

Attributed term induction, developed as a proof principle for attributed algebraic specifications [Bauer 94a, 94b, 95], can be used to prove properties between attributes of occurrence terms of a distinguished sort. Attributed term induction is a special case for computing the complete set of occurrence terms of a distinguished sort.

The idea of the proof principle is to split an occurrence term into a subordinate and a superior term as visualized in figure 19. Two kinds of induction are combined for building the superior term (a special kind of context induction with another ordering) and for building the subordinate term (a special kind of term induction).

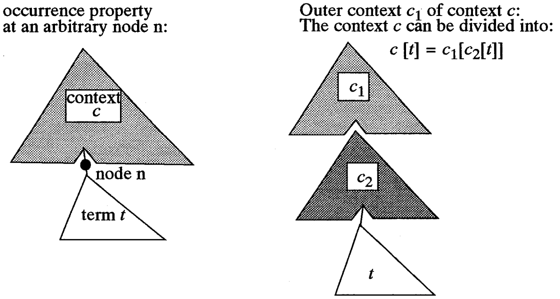


figure 19: splitting the abstract syntax tree and the notion of an outer context

Since an occurrence term can be written as $c[occ(t)]$, it is sufficient to show that

$$\forall\, c[z_{s_t}] \in (T_{(S,\, C,\, \varnothing)}(\{\, z_{s_t} \,\}))_{s_{root}}.\ \forall\, t \in (T_{(S,\, C,\, \varnothing)})_{s_t}.\ P(c[\mathbf{occ}_s(t)])$$

is valid and P is a property for occurrence terms of sort s_t and root sort s_{root} (called *occurrence property*).

The notion of an outer context is needed for the definition of the attributed term induction which is the induction ordering on contexts:

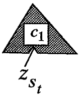
Definition 5.2.2.1 (outer context)

$c_1[z_s]$ is an *outer context* of $c[z_s]$ if there exists a (non-trivial) context $c_2[z_s]$ such that $c[z_s] \equiv c_1[c_2[z_s]]$. More graphically speaking on the path from the root of $c_1[z_s]$ to the context identifier z_s the number of nodes of sort s is smaller than the number of nodes of sort s on the path from the root of $c[z_s]$ to the context identifier z_s. The notion of an outer context defines a Noetherian relation on contexts.

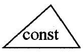
The attributed term induction can be explained in an abstract way considering the case analyses which have to be performed. At the outermost level a kind of context induction [Hennicker 91] with the same notion of context, but with another ordering, and at the innermost level term induction (cf. e.g. [Ehrich et al. 89]) is performed:

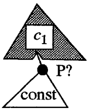
(1) Base of the context induction:

In this case a context with minimal insertion place of sort s_t has to be considered, i.e. the set of minimal outer contexts.

Let  be such minimal contexts for terms of sort s_t. A term induction has to be performed for this sort.

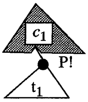
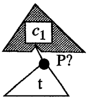
(1.1) Base of the term induction:

Let  denote all constants of appropriate sort s_t. The proof obligation is the occurrence property at the node marked with *P?* in the abstract term[9]:



P? denotes that at this node the occurrence property has to be valid.

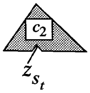
(1.2) Induction step of the term induction:

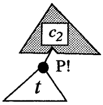
The induction assertion is  (*P!* denotes that at this node the occurrence property is valid) such that t_1 is any subterm of sort s_t of the term t of the proof obligation  .

[9] Note, that the abstract term is an abbreviation for a set of terms.

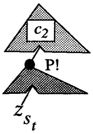
(2) Context induction step:

In (1) the property has been shown for contexts with minimal insertion place depth. The induction step starts with an insertion place of depth n for which the property is valid (induction assertion) and considers an insertion place of depth $n + 1$ for which the occurrence property has to be proved.

Let 

be a given outer context for terms of sort s_t with insertion place of depth n. It is abstracted from the concrete representation of the context, i.e. an arbitrary context $c_2[z_{s_t}]$ of depth n is considered. The induction assertion is 

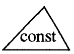
for arbitrary terms t of sort s_t. Since in (1) we have shown that the property is valid for all minimal contexts and for all terms.

Let 

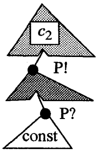
denote contexts for terms of sort s_t with insertion place of depth $n + 1$.

Now a term induction for this sort is performed.

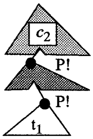
(2.1) Base of the term induction:

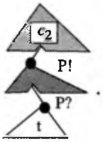
Let 

denote all constants of appropriate sort. In the proof obligation the following term have to be considered:



(2.2) Induction step of the term induction:

The induction assertion is 

such that t_1 is a subterm with sort s_t of the term t of the proof obligation  .

This finishes the proof. ♦

As already stated an occurrence term $t^{occ_s} \in T^{occ_s}_{(S,\,C,\,\varnothing)}$ can be written as $c[\mathbf{occ}_s(t)]$ for some context $c[z_s]$ and some term $t \in (T_{(S,\,C,\,\varnothing)})_s$ leading to the following proof principle:

Definition 5.2.2.2 (Attributed Term Induction):

Let $ASpec = <\Sigma, F_{Attr}, Ax>$ with $\Sigma = (S, C, F)$ be an attributed algebraic specification and P the occurrence property for occurrence terms of sort s_t.

To show that P holds for all occurrence terms of sort s_t, it is sufficient to show[10]:

(1) It holds for all minimal outer contexts $c_1[z_{s_t}]$ of sort s_{root}:

(1.1) $P(c_1[\mathbf{occ}(f)])$ is valid for all constants $f \in C$ of sort s_t;

(1.2) It holds for all terms $f(t_1, t_2,..., t_n)$ with constructor symbol $(f\colon s_1, s_2,..., s_n \rightarrow s_t) \in C$ and terms $t_i \in (T_{(S,\,C,\,\varnothing)})_{s_i}$ $(1 \leq i \leq n)$:
Under the assumption that $P(c_1[\mathbf{occ}(t')])$ is valid for all subterms t' of $f(t_1, t_2,..., t_n)$ of sort s_t, $P(c_1[\mathbf{occ}(f(t_1, t_2,..., t_n))])$ must be valid;
In particular, the validity of $P(c_1[\mathbf{occ}(t')])$ can be assumed, if $sort(t') = s_t$;

(2) and

(2.1) Under the assumption that $P(c_2[\mathbf{occ}(t)])$ is valid for all outer contexts $c_2[z_{s_t}]$ of a context c_3 and all terms $t \in (T_{(S,\,C,\,\varnothing)})_{s_t}$, $P(c_3[\mathbf{occ}(f)])$ must be valid for all constants $f \in C$ of sort s_t;
In particular, the validity of $P(c_2[\mathbf{occ}(t)])$ can be assumed if $t \in (T_{(S,\,C,\,\varnothing)})_{s_t}$;

(2.2) It holds for all terms $f(t_1, t_2,..., t_n)$ with constructor symbol $(f\colon s_1, s_2,..., s_n \rightarrow s_t) \in C$ and terms $t_i \in (T_{(S,\,C,\,\varnothing)})_{s_i}$ $(1 \leq i \leq n)$:
Under the assumption that $P(c_2[\mathbf{occ}(t)])$ is valid for all outer contexts $c_2[z_{s_t}]$ of a context c_3 and all terms $t \in (T_{(S,\,C,\,\varnothing)})_{s_t}$ and $P(c_4[\mathbf{occ}(t')])$ is valid for all subterms t' of $f(t_1, t_2,..., t_n)$ of appropriate sort and all contexts c_4, $P(c_3[\mathbf{occ}(f(t_1, t_2,..., t_n))])$ must be valid;
In particular, the validity of $P(c_2[\mathbf{occ}(t)])$ can be assumed if $t \in (T_{(S,\,C,\,\varnothing)})_{s_t}$ and the validity of $P(c_4[\mathbf{occ}(t')])$ can be assumed, if $sort(t') = s_t$;

Theorem 5.2.2.3 (Soundness of Attributed Term Induction):

The proof principle of attributed term induction is sound. ♦

[10] It is assumed that there exists only one root sort s_{root}, i.e. all attribute functions have the functionality $fct(f_{Attr}) = s_{root} \rightarrow s$ for all $f_{Attr} \in F_{Attr}$ with arbitrary $s \in S$.

Sketch of the Proof:

It is obvious that the outer context relation defines a Noetherian relation using the ordering on natural numbers and the depth of the insertion place.

The proof obligation, which has to be shown by the proof principle, is:

$$\forall\, c[z_{s_t}] \in (T_{(S,\,C,\,\varnothing)}(\{z_{s_t}\}))_{s_{root}}.\ \forall\, t \in (T_{(S,\,C,\,\varnothing)})_{s_t}.\ P(c[\mathbf{occ}(t)])$$

which can be rewritten as

$$\forall\, c[z_{s_t}] \in (T_{(S,\,C,\,\varnothing)}(z_{s_t}))_{s_{root}}.\ P_{c[z_{s_t}]}$$

with

$$P_{c[z_{s_t}]} = \forall\, t \in (T_{(S,\,C,\,\varnothing)})_{s_t}.\ P(c[\mathbf{occ}(t)])$$

The notion of an outer context defines a Noetherian relation on all contexts $c[z_{s_t}] \in (T_{(S,\,C,\,\varnothing)}(\{z_{s_t}\}))_{s_{root}}$. Therefore Noetherian induction can be used, leading to the proof obligations:

(1) $P_{c[z_{s_t}]}$ has to be valid for all minimal outer contexts.

(2) under the assumption that $P_{c'[z_{s_t}]}$ is valid for all outer context $c'[z_{s_t}]$ of $c[z_{s_t}]$, $P_{c[z_{s_t}]}$ has to be valid.

$P_{c[z_{s_t}]} = \forall\, t \in (T_{(S,\,C,\,\varnothing)})_{s_t}.\ P(c[\mathbf{occ}(t)])$ can be shown by term induction.

Viewing the above proof obligation in this way the soundness of the proof principle follows immediately. ♦

The proof principle of attributed term induction was implemented in [Duschl 94; Weiß 95] generating proof obligations which are shown by the TIP system [Fraus 94a, 94b].

Example 5.2.2.4

The example is based on an implementation of the specification *CMOBILE*. The specification *CMOBILE*1 is defined as:

```
aspec CMOBILE1 =
  enrich NAT by
    sorts Mobile
    cons mobile: Mobile, Mobile → Mobile,
           cube: Nat → Mobile,
           fixed: → Nat
    attrs   synth   weight, leftlength, rightlength, cmaxdepth: Mobile → Nat
            inh     length, depth, maxdepth: Mobile → Nat
    axioms for all sv: Mobile → Mobile; m, m1, m2: Mobile; l: Nat.
    (1)    weight(sv[occ(cube(l))]) = l,
    (2)    weight(sv[occ(mobile(m1, m2))]) =
              weight(sv[mobile(occ(m1), m2)]) + weight(sv[mobile(m1, occ(m2))]),

    (3)    length(sv[occ(cube(l)]) = 0,

    (4)    leftlength(sv[occ(mobile(m1, m2))]) = length(sv[occ(mobile(m1, m2))]) *
              weight(sv[mobile(m1, occ(m2))]) / weight(sv[occ(mobile(m1, m2))]),
```

(5) rightlength(sv[**occ**(mobile(m1, m2))]) = length(sv[**occ**(mobile(m1, m2))]) *
weight(sv[mobile(**occ**(m1), m2)]) / weight(sv[**occ**(mobile(m1, m2))]),
(6) leftlength(sv[**occ**(cube(l))]) = 0,
(7) rightlength(sv[**occ**(cube(l))]) = 0,

(8) depth(**occ**(m)) = 1,
(9) depth(sv[mobile(**occ**(m1), m2)]) = depth(sv[**occ**(mobile(m1, m2))]) + 1,
(10) depth(sv[mobile(m1, **occ**(m2))]) = depth(sv[**occ**(mobile(m1, m2))]) + 1,
(11) cmaxdepth(sv[**occ**(cube(l))]) = depth(sv[**occ**(cube(l))]),
(12) cmaxdepth(sv[**occ**(mobile(m1, m2))]) =
max(cmaxdepth(sv[mobile(**occ**(m1), m2)]), cmaxdepth(sv[mobile(m1, **occ**(m2))])),

(13) length(**occ**(m)) = fixed * maxdepth(**occ**(m)),
(14) length(sv[mobile(**occ**(m1), m2)]) = length(sv[**occ**(mobile(m1, m2))]) - fixed,
(15) length(sv[mobile(m1, **occ**(m2))]) = length(sv[**occ**(mobile(m1, m2))]) - fixed,

(16) maxdepth(**occ**(m)) = cmaxdepth(**occ**(m)),
(17) maxdepth(sv[mobile(**occ**(m1), m2)]) = maxdepth(sv[**occ**(mobile(m1, m2))]),
(18) maxdepth(sv[mobile(m1, **occ**(m2))]) = maxdepth(sv[**occ**(mobile(m1, m2))])

endspec

The specification is a two-pass attribution without remote access of the attribute values. The following property is necessary for the proof of the implementation relation:

It holds all nodes, i.e. for all occurrence terms of sort *Mobile*:

The sum of the attribute occurrences of left length and right length of a submobile is equal to the attribute occurrence of the maximal depth of the mobile minus the actual depth plus one multiplied with the constant fixed, or the sum of the left length and right length of a submobile is zero.

Formally:

$\forall\, sv_{Mobile \rightarrow Mobile}.\ \forall\, m_{Mobile}.$

$$ll(sv[\mathbf{occ}(m)] + rl(sv[\mathbf{occ}(m)]) = fixed * (md(sv[\mathbf{occ}(m)]) - d(sv[\mathbf{occ}(m)]) + 1)$$

v

$$ll(sv[\mathbf{occ}(m)]) + rl(sv[\mathbf{occ}(m)]) = 0$$

Denoted by $P(sv[\mathbf{occ}(m)])$.

(Using the abbreviations *sv*, *m*, *l*, *ll*, *rl*, *d*, *md* and *cmd* for $sv_{Mobile \rightarrow Mobile}$, m_{Mobile}, *length*, *leftlength*, *rightlength*, *depth*, *maxdepth* and *cmaxdepth*, respectively.)

Informally we can say:

> The length (= *ll* + *rl*) of a mobile is
> - at the top level *fixed* * *md*,
> - at the next level *fixed* * (*md* - 1)
> and so on.

This property can be shown by attributed term induction:
(1) The minimal contexts to consider is: z_{Mobile}
(1.1) No constants of sort *Mobile* exist. Therefore nothing is to show.
(1.2) The constructor symbols to consider are *cube* and *mobile*. Thus the occurrence

terms are
$\mathbf{occ}(cube(l))$ for an arbitrary term l of sort *Nat*
and
$\mathbf{occ}(mobile(m1, m2))$ for arbitrary terms $m1$ and $m2$ of sort *Mobile*.
$P(\mathbf{occ}(cube(l)))$ is valid since $ll(\mathbf{occ}(cube(l)))$ and $rl(\mathbf{occ}(cube(l)))$ are both zero.
Using the abbreviations:
$ll_0 = ll(\mathbf{occ}(mobile(m1, m2)))$, $rl_0 = rl(\mathbf{occ}(mobile(m1, m2)))$,
$l_0 = l(\mathbf{occ}(mobile(m1, m2)))$
$md_0 = md(\mathbf{occ}(mobile(m1, m2)))$, $d_0 = d(\mathbf{occ}(mobile(m1, m2)))$,
$w_1 = w(mobile(\mathbf{occ}(m1), m2))$, $w_2 = w(mobile(m1, \mathbf{occ}(m2)))$
and the informations from the specification:
$d_0 = 1$, $w_0 = w_1 + w_2$, $ll_0 = l_0 * (w_2 / w_0)$, $rl_0 = l_0 * (w_1 / w_0)$, $l_0 = fixed * md_0$
it holds:
$ll_0 + rl_0 = l_0 * (w_2 / w_0) + l_0 * (w_1 / w_0) = l_0 = fixed * md_0 = fixed * (md_0 - 1 + 1) = fixed * (md_0 - d_0 + 1)$

(2) The contexts to consider are: $c[mobile(z_{Mobile}, m)]$ and $c[mobile(m, z_{Mobile})]$.
The proof is only given for the context $c[mobile(z_{Mobile}, m)]$, the proof for the context $c[mobile(m, z_{Mobile})]$ is analogous.
The induction assertion is for the considered case:
$P(c[\mathbf{occ}(mobile(m1, m2))])$ for arbitrary terms $m1$ and $m2$ of sort *Mobile*.

(2.1) No constants of sort *Mobile* exist. Therefore nothing is to show.

(2.2) The constructor symbols to consider are *cube* and *mobile*. Thus the occurrence terms are
$c[mobile(\mathbf{occ}(cube(l)), m)]$ for an arbitrary term l of sort *Nat*.
and
$c[mobile(\mathbf{occ}(mobile(m1, m2)), m)]$ for arbitrary terms $m1$ and $m2$ of sort *Mobile*.
$P(c[mobile(\mathbf{occ}(cube(l)), m)])$ is valid,
since $ll(c[mobile(\mathbf{occ}(cube(l)), m)])$ and $rl(c[mobile(\mathbf{occ}(cube(l)), m)])$ are both zero.
Using the abbreviations:
$ll_0 = ll(c[\mathbf{occ}(mobile(mobile(m1, m2), m)])$,
$rl_0 = rl(c[\mathbf{occ}(mobile(mobile(m1, m2), m)])$,
$l_0 = l(c[\mathbf{occ}(mobile(mobile(m1, m2), m)])$,
$w_0 = w(c[\mathbf{occ}(mobile(mobile(m1, m2), m)])$
$ll_1 = ll(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$rl_1 = rl(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$l_1 = l(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$md_1 = md(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$d_1 = d(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$d_2 = d(c[mobile(mobile(m1, m2), \mathbf{occ}(m))])$
$w_1 = w(c[mobile(\mathbf{occ}(mobile(m1, m2)), m)])$,
$w_2 = w(c[mobile(mobile(m1, m2), \mathbf{occ}(m))])$,
$w_{11} = w(c[mobile(mobile(\mathbf{occ}(m1), m2), m)])$,
$w_{12} = w(c[mobile(mobile(m1, \mathbf{occ}(m2)), m)])$
the induction assertion
$ll_0 + rl_0 = fixed * (md_0 - d_0 + 1)$

and the informations from the specification:
$w_0 = w_1 + w_2$, $w_1 = w_{11} + w_{12}$, $d_1 = d_0 + 1$
$ll_1 = l_1 * (w_{12} / w_1)$, $rl_1 = l_1 * (w_{11} / w_1)$, $l_1 = l_0 - \mathit{fixed}$,
$ll_0 = l_0 * (w_2 / w_0)$, $rl_0 = l_0 * (w_1 / w_0)$
it holds:
It follows from the induction assertion:
$ll_0 + rl_0 = \mathit{fixed} * (md_0 - d_0 + 1)$ iff $l_0 * (w_2 / w_0) + l_0 * (w_1 / w_0) =$
$\mathit{fixed} * (md_0 - d_0 + 1)$ iff
$l_0 * ((w_2 + w_1) / w_0) = \mathit{fixed} * (md_0 - d_0 + 1)$ iff $l_0 = \mathit{fixed} * (md_0 - d_0 + 1)$
lhs:
$ll_1 + rl_1 = l_1 * (w_{12} / w_1) + l_1 * (w_{11} / w_1) = l_1 = l_0 - \mathit{fixed} =$
$\mathit{fixed} * (md_0 - d_0 + 1) - \mathit{fixed} = \mathit{fixed} * (md_0 - d_0)$
rhs:
$\mathit{fixed} * (md_1 - d_1 + 1) = \mathit{fixed} * (md_0 - d_0 -1 + 1) = \mathit{fixed} * (md_0 - d_0)$ ♦

This example shows that as well the context induction as the term induction is necessary to verify the proof obligation.

5.2.3 Induction Orderings

Attributed term induction combines two Noetherian induction principles. In this section we define an induction ordering, i.e. a Noetherian and stable partial ordering, on occurrence terms, which is compatible with the ordering used in attributed term induction.

This ordering in connection with a complete set of occurrence terms is a generalization of attributed term induction, which defines a special complete set of occurrence terms and applies explicit induction assertions.

The correctness of the generalized proof principle implies the correctness of attributed term induction.

The notions of orderings and inductions are defined as usual:

A partial order is defined as follows (cf. [Reade 89]):

Definition 5.2.3.1 (partial order, well founded, Noetherian):

A binary relation $< \subseteq S \times S$ for an arbitrary set S is called *partial order*, if $<$ satisfies the following two properties:

(1) transitivity: for all $x, y, z \in S$ holds: $x < y$ and $y < z$ implies $x < z$.
(2) anti-symmetry: for all $x, y \in S$ holds: $x < y$ implies that not $y < x$.

A partial order $(S, <)$ is said to be *well founded* or *Noetherian*, if there are no infinite decreasing chains, i.e. there are no infinite sequences of elements from S, i.e. $\{x_0, x_1, ...\}$ such that $x_{i+1} < x_i$ for all $i \geq 0$. ♦

Definition 5.2.3.2 (complete induction)

The principle of *complete induction* say that if $(S, <)$ is a well founded partial order, and $P(s)$ is a property about elements $s \in S$, then in order to show that $P(s)$ is true for all $s \in S$ it is sufficient to show (for any $s \in S$):

$P(s)$ follows from the assumption that $P(t)$ is true for all $t < s$.

The assumption $P(t)$ for all $t < s$ is called the *induction assertion*. There may be $s \in S$ with no $t < s$. For such elements the principle says that we must show $P(s)$ with no assumptions. ♦

Having a complete set of occurrence terms with subterm identifiers the notion of a *stable ordering* is necessary to get an induction ordering:

Definition 5.2.3.3 (stable ordering)

A partial ordering $< \subseteq S \times S$ is called *stable* iff it is closed under substitution. ♦

Definition 5.2.3.4 (induction ordering)

Every Noetherian and stable partial ordering is an *induction ordering*. ♦

The aim is the definition of an induction ordering on occurrence terms:

Definition 5.2.3.5 (induction ordering on occurrence terms)

The relation $t <_{occ_s} r$ on occurrence terms $t, r \in T^{occ_s}_{(S,\, C,\, \varnothing)}(SV)$ is defined as:
$t <_{occ_s} r$ iff

(1) $t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[\mathbf{occ}_s(c'[t'])]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t'

or[11]

(2) $t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[c'[\mathbf{occ}_s(r')]]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t' and r'. ♦

To prove that $<_{occ_s}$ is an induction ordering it has to be a partial ordering.

Lemma 5.2.3.6

$<_{occ_s}$ is a partial ordering on occurrence terms $T^{occ_s}_{(S,\, C,\, \varnothing)}(SV)$.

Proof

Transitivity:

Let $t_1, t_2, t_3 \in T^{occ_s}_{(S,\, C,\, \varnothing)}(SV)$ be arbitrary such that $t_1 <_{occ_s} t_2$ and $t_2 <_{occ_s} t_3$.

$t_1 <_{occ_s} t_2$ holds if

$t_1 \equiv c_1[\mathbf{occ}_s(t_1')]$ and $t_2 \equiv c_1[\mathbf{occ}_s(c_1'[t_1'])]$ for some contexts $c_1[z_s]$, $c_1'[z_s] \not\equiv z_s$ and term t_1'

or

$t_1 \equiv c_2[\mathbf{occ}_s(t_2')]$ and $t_2 \equiv c_2[c_2'[\mathbf{occ}_s(t_3')]]$ for some contexts $c_2[z_s]$, $c_2'[z_s] \not\equiv z_s$ and term t_2' and t_3'

and

[11] Because of the condition $c_1'[z_s] \not\equiv z_s$ in (2) the „or“ can be read as „either-or“.

$t_2 <_{occ_s} t_3$ holds if

$t_2 \equiv c_3[\mathbf{occ}_s(t_4')]$ and $t_3 \equiv c_3[\mathbf{occ}_s(c_3'[t_4'])]$ for some contexts $c_3[z_s]$, $c_3'[z_s] \not\equiv z_s$ and term t_4'

or

$t_2 \equiv c_4[\mathbf{occ}_s(t_5')]$ and $t_3 \equiv c_4[c_4'[\mathbf{occ}_s(t_6')]]$ for some contexts $c_4[z_s]$, $c_4'[z_s] \not\equiv z_s$ and terms t_5', t_6'

is equivalent to (with the same restrictions)[12]

$t_1 \equiv c_1[\mathbf{occ}_s(t_1')]$ and $t_2 \equiv c_1[\mathbf{occ}_s(c_1'[t_1'])]$ and $t_2 \equiv c_3[\mathbf{occ}_s(t_4')]$ and $t_3 \equiv c_3[\mathbf{occ}_s(c_3'[t_4'])]$

or

$t_1 \equiv c_2[\mathbf{occ}_s(t_2')]$ and $t_2 \equiv c_2[c_2'[\mathbf{occ}_s(t_3')]]$ and $t_2 \equiv c_3[\mathbf{occ}_s(t_4')]$ and $t_3 \equiv c_3[\mathbf{occ}_s(c_3'[t_4'])]$

or

$t_1 \equiv c_1[\mathbf{occ}_s(t_1')]$ and $t_2 \equiv c_1[\mathbf{occ}_s(c_1'[t_1'])]$ and $t_2 \equiv c_4[\mathbf{occ}_s(t_5')]$ and $t_3 \equiv c_4[c_4'[\mathbf{occ}_s(t_6')]]$

or

$t_1 \equiv c_2[\mathbf{occ}_s(t_2')]$ and $t_2 \equiv c_2[c_2'[\mathbf{occ}_s(t_3')]]$ and $t_2 \equiv c_4[\mathbf{occ}_s(t_5')]$ and $t_3 \equiv c_4[c_4'[\mathbf{occ}_s(t_6')]]$

implies

$t_1 \equiv c_1[\mathbf{occ}_s(t_1')]$ and $t_2 \equiv c_1[\mathbf{occ}_s(c_1'[t_1'])]$ and $t_2 \equiv c_3[\mathbf{occ}_s(t_4')]$ and $t_3 \equiv c_1[\mathbf{occ}_s(c_3'[c_1'[t_1']])]$

or

$t_1 \equiv c_2[\mathbf{occ}_s(t_2')]$ and $t_2 \equiv c_2[c_2'[\mathbf{occ}_s(t_3')]]$ and $t_2 \equiv c_3[\mathbf{occ}_s(t_4')]$ and $t_3 \equiv c_2[c_2'[\mathbf{occ}_s(c_3'[t_4'])]]$

or

$t_1 \equiv c_1[\mathbf{occ}_s(t_1')]$ and $t_2 \equiv c_1[\mathbf{occ}_s(c_1'[t_1'])]$ and $t_2 \equiv c_4[\mathbf{occ}_s(t_5')]$ and $t_3 \equiv c_1[c_4'[\mathbf{occ}_s(t_6')]]$

or

$t_1 \equiv c_2[\mathbf{occ}_s(t_2')]$ and $t_2 \equiv c_2[c_2'[\mathbf{occ}_s(t_3')]]$ and $t_2 \equiv c_4[\mathbf{occ}_s(t_5')]$ and $t_3 \equiv c_2[c_2'[c_4'[\mathbf{occ}_s(t_6')]]]$

implies

$t_1 <_{occ_s} t_3$ or $t_1 <_{occ_s} t_3$ or $t_1 <_{occ_s} t_3$ or $t_1 <_{occ_s} t_3$

implies

$t_1 <_{occ_s} t_3$

Anti-Symmetric:

To show: $t <_{occ_s} r$ does not imply $r <_{occ_s} t$

Let $t <_{occ_s} r$, i.e.

$t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[\mathbf{occ}_s(c'[t'])]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t', which implies that (1) and (2) of the definition of $r <_{occ_s} t$ is not valid, since otherwise

[12] $(a \vee b) \wedge (c \vee d) = ((a \vee b) \wedge c) \vee ((a \vee b) \wedge d) = (a \wedge c) \vee (b \wedge c) \vee (a \wedge d) \vee (b \wedge d)$

$c'[z_s] \equiv z_s$ has to be valid, being a constradiction to the assumption

or

$t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[c'[\mathbf{occ}_s(r')]]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t' and r', which implies that (1) and (2) of the definition of $r <_{occ_s} t$ is not valid, since otherwise $c'[z_s] \equiv z_s$ has to be valid, being a constradiction to the assumption. ♦

As a next step we show that the ordering $<_{occ_s}$ is closed under substitution.

Lemma 5.2.3.7

$<_{occ_s}$ is closed under substitution, i.e. under the assumption $t <_{occ_s} r$ it must be shown for all $\sigma \in \mathit{Subst}$:

$\sigma(t) <_{occ_s} \sigma(r)$. ♦

Proof

Follows immediately from the inductive definition of the substitution:

Let $t <_{occ_s} r$, i.e.

$t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[\mathbf{occ}_s(c'[t'])]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t'

or

$t \equiv c[\mathbf{occ}_s(t')]$ and $r \equiv c[c'[\mathbf{occ}_s(r')]]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t' and r'

implies

$\sigma(t) \equiv \sigma(c[\mathbf{occ}_s(t')])$ and $\sigma(r) \equiv \sigma(c[\mathbf{occ}_s(c'[t'])])$ for some contexts $\sigma(c[z_s])$, $\sigma(c'[z_s]) \not\equiv z_s$ and term $\sigma(t')$

or

$\sigma(t) \equiv \sigma(c[\mathbf{occ}_s(t')])$ and $\sigma(r) \equiv \sigma(c[c'[\mathbf{occ}_s(r')]])$ for some contexts $\sigma(c[z_s])$, $\sigma(c'[z_s]) \not\equiv z_s$ and term $\sigma(t')$ and $\sigma(r')$

and therefore $\sigma(t) <_{occ_s} \sigma(r)$ is valid. ♦

Lemma 5.2.3.8

$<_{occ_s}$ is a well founded ordering, i.e. a Noetherian ordering. ♦

Proof

In order to show that there are no decreasing infinite chains wrt. $<_{occ_s}$, we define the set of minimal elements wrt. $<_{occ_s}$ and show that no ordering exists on this set of minimal elements:

The minimal set MS of occurrence terms $t \in T^{occ_s}_{(S,C,\varnothing)}(SV)$ is defined as

$MS = \{\ c[\mathbf{occ}_s(t)] \mid c[z_s] \not\equiv c_1[c_2[z_s]]$ with arbitrary contexts $c_1[z_s] \not\equiv z_s$ and $c_2[z_s] \not\equiv z_s$ and $t \not\equiv c_3[t']$ with context $c_3[z_s]$ and term t' of sort $s\ \}$

All *ms* ∈*MS* are minimal:

Let *ms* ∈*MS* be arbitrary. We assume that there exists an occurrence term $t \in (T^{occ_s}_{(S,C,\varnothing)}(SV)_{s'})$ such that $t <_{occ_s} ms$. Since *ms* is an occurrence term it can be written as

$c[\mathbf{occ}_s(r)]$ for some context $c[z_s]$ and term r.

If $t <_{occ_s} ms$ is valid, t must have the form

$t \equiv c[\mathbf{occ}_s(t')]$ with $r \equiv c'[t']$ for some context $c'[z_s]$
which is a contradiction to *ms* ∈*MS*: since $c[\mathbf{occ}_s(r)]$ implies that r cannot be written as $c'[t']$ for some context $c'[z_s]$.

or t must have the form

$t \equiv c_1[\mathbf{occ}_s(t')]$ where $c[z_s]$ can be written as $c_1[c_2[z_s]]$ with arbitrary contexts $c_1[z_s]$ and $c_2[z_s] \not\equiv z_s$, which is again a contradiction to *ms* ∈*MS*: since $c[\mathbf{occ}_s(r)]$ implies that $c[z_s]$ cannot be written as $c_1[c_2[z_s]]$ with arbitrary contexts $c_1[z_s]$ and $c_2[z_s] \not\equiv z_s$.

Therefore all *ms* ∈*MS* are minimal.

No ordering is defined on the elements of *MS*:

Let ms_1, ms_2 ∈*MS* be arbitrary elements such that $ms_1 \equiv c_1[\mathbf{occ}_s(t_1)]$ and $ms_2 \equiv c_2[\mathbf{occ}_s(t_2)]$ with the restriction defined in *MS*. We have to show that neither $ms_1 <_{occ_s} ms_2$ nor $ms_2 <_{occ_s} ms_1$ holds:

Assume $ms_1 <_{occ_s} ms_2$ then

$ms_1 \equiv c[\mathbf{occ}_s(t')]$ and $ms_2 \equiv c[\mathbf{occ}_s(c'[t'])]$ for some contexts $c'[z_s] \not\equiv z_s$, $c'[z_s] \not\equiv z_s$ and term t'. But this is a contradiction to the fact that t_2 cannot be written as $c'[t']$ with $c'[z_s] \not\equiv z_s$,

or

$ms_1 \equiv c[\mathbf{occ}_s(t')]$ and $ms_2 \equiv c[c'[\mathbf{occ}_s(r')]]$ for some contexts $c[z_s]$, $c'[z_s] \not\equiv z_s$ and term t' and r'. But this is a contradiction to the fact that $c_1[z_s]$ cannot be written as $c[c'[z_s]]$.

$ms_1 <_{occ_s} ms_2$ can be shown analogously ◆

Corollary 5.2.3.9

$<_{occ_s}$ is an induction ordering ◆

Corollary 5.2.3.10

To show $\forall\, t \in T_\Sigma^{occ_s}(SV)$. $P(t)$ it is sufficient to prove

$\forall\, t \in T_\Sigma^{occs}(SV)$. ($P(t')$ and $t' <_{occ_s} t$) implies $P(t)$

for some property P on occurrence terms t. ◆

Proof

The correctness follows from the fact that $<_{occ_s}$ is an induction ordering. ♦

Sometimes a property describes dependencies between occurrence terms of maybe different sorts. An induction ordering can be defined for these properties.

Corollary 5.2.3.11 (induction ordering on different occurrence terms)

Let $[t_1, t_2,\ldots, t_n]$ and $[r_1, r_2,\ldots, r_n]$ be lists of different occurrence terms. The following lexicographical extension $<_{Occ}$ of $<_{occ_s}$, defined as

$$[t_1, t_2,\ldots, t_n] <_{Occ} [r_1, r_2,\ldots, r_n] \text{ if } t_j \equiv r_j \text{ holds for all } 1 \leq j < i \leq n \text{ and } t_i <_{occ_s} r_i$$

is an induction ordering on different occurrence terms. ♦

Proof

The correctness immediately follows from the fact that $<_{occ_s}$ is an induction ordering and therefore its lexicographic extension $<_{Occ}$ is a partial ordering, closed under substitution and Noetherian. ♦

Definition 5.2.3.12 (complete set for a subterm identifier)

$CS_{sv} \subseteq T_{(S,\,C,\,\varnothing)}(SV)$ is called *complete set for a subterm identifier* $sv_{s1,\,s2,\ldots,\,s_n \to s}$ iff

$$\forall\, c[z_{s_1}, z_{s_2},\ldots, z_{s_n}] \in T_{(S,\,C,\,\varnothing)}(\{\, z_{s_1}, z_{s_2},\ldots, z_{s_n} \,\}).\ \exists\, cs \in CS_{sv}.\ \exists\, \sigma \in Subst.$$
$$\sigma(cs) = c[z_{s_1}, z_{s_2},\ldots, z_{s_n}]$$

Especially the property is written for $n = 0$ as:

$$\forall\, t \in T_{(S,\,C,\,\varnothing)}.\ \exists\, cs \in CS_{sv}.\ \exists\, \sigma \in Subst.\ \sigma(cs) = t$$

which is the definition of a complete set for usual identifiers. ♦

Example 5.2.3.13

Let us consider the same example as for the attributed term induction with the differences that a remote access to attribute occurrences is used and the proof is performed with complete sets:

$\forall\, sv_{Mobile \to Mobile}.\ \forall\, m_{Mobile}.$

$ll(sv[\mathbf{occ}(m)] + rl(sv[\mathbf{occ}(m)]) = fixed * (cmd(\mathbf{occ}(sv[m])) - d(sv[\mathbf{occ}(m)]) + 1)$

$\vee\ ll(sv[\mathbf{occ}(m)]) + rl(sv[\mathbf{occ}(m)]) = 0$

Denoted by $P(sv[\mathbf{occ}(m)], \mathbf{occ}(sv[m]))$.

The complete sets we consider are

$$CS_{sv} = \{\, z_{Mobile}, sv[mobile(z_{Mobile}, m)], sv[mobile(m, z_{Mobile})] \,\}$$

and

$$CS_m = \{\, cube(l), mobile(m1, m2) \,\}$$

The proof obligations are

(1) $P(\mathbf{occ}(\mathit{cube}(l)), \mathbf{occ}(\mathit{cube}(l)))$,

(2) $P(\mathbf{occ}(\mathit{mobile}(m1, m2)), \mathbf{occ}(\mathit{mobile}(m1, m2)))$

(3) $P(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l)), m)], \mathbf{occ}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l), m)]))$

(4) $P(\mathit{sv}[\mathit{mobile}(m, \mathbf{occ}(\mathit{cube}(l)))], \mathbf{occ}(\mathit{sv}[\mathit{mobile}(\mathrm{m}, \mathit{cube}(l))]))$

(5) $P(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)], \mathbf{occ}(\mathit{sv}[\mathit{mobile}(\mathit{mobile}(m1, m2), m)]))$

(6) $P(\mathit{sv}[\mathit{mobile}(\mathrm{m}, \mathbf{occ}(\mathit{mobile}(m1, m2)))], \mathbf{occ}(\mathit{sv}[\mathit{mobile}(m, \mathit{mobile}(m1, m2))]))$

proof of (1)

$P(\mathbf{occ}(\mathit{cube}(l)), \mathbf{occ}(\mathit{cube}(l)))$ is valid since $\mathit{ll}(\mathbf{occ}(\mathit{cube}(l)))$ and $\mathit{rl}(\mathbf{occ}(\mathit{cube}(l)))$ are both zero.

proof of (2)

Using the abbreviations

$\mathit{ll}_0 = \mathit{ll}(\mathbf{occ}(\mathit{mobile}(m1, m2)))$, $\mathit{rl}_0 = \mathit{rl}(\mathbf{occ}(\mathit{mobile}(m1, m2)))$, $l_0 = l(\mathbf{occ}(\mathit{mobile}(m1, m2)))$
$\mathit{md}_0 = \mathit{md}(\mathbf{occ}(\mathit{mobile}(m1, m2)))$, $d_0 = d(\mathbf{occ}(\mathit{mobile}(m1, m2)))$,
$w_1 = w(\mathit{mobile}(\mathbf{occ}(m1), m2))$, $w_2 = w(\mathit{mobile}(m1, \mathbf{occ}(m2)))$

and the axioms of the specification:
$d_0 = 1$, $w_0 = w_1 + w_2$, $\mathit{ll}_0 = l_0 * (w_2 / w_0)$, $\mathit{rl}_0 = l_0 * (w_1 / w_0)$, $l_0 = \mathit{fixed} * \mathit{md}_0$

it holds:
$\mathit{ll}_0 + \mathit{rl}_0 = l_0 * (w_2 / w_0) + l_0 * (w_1 / w_0) = l_0 = \mathit{fixed} * \mathit{md}_0 = \mathit{fixed} * (\mathit{md}_0 - 1 + 1) = \mathit{fixed} * (\mathit{md}_0 - d_0 + 1)$

proof of (3)

$\mathit{ll}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{cube}(l)), m)])$ and $\mathit{rl}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{cube}(l)), m)])$ are both zero.

proof of (4)

analogous to (3).

proof of (5)

The induction assertion is for the considered case:
$P(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))], \mathbf{occ}(\mathit{sv}[\mathit{mobile}(m1, m2)))])$ for identifiers $m1$ and $m2$ of sort *Mobile*.

Using the abbreviations:
$\mathit{md} = \mathit{md}(\mathbf{occ}(\mathit{sv}[\mathit{mobile}(\mathit{mobile}(m1, m2), m)]))$,
$\mathit{ll}_0 = \mathit{ll}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(\mathit{mobile}(m1, m2), m)])$, $\mathit{rl}_0 = \mathit{rl}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(\mathit{mobile}(m1, m2), m)])$,
$l_0 = l(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(\mathit{mobile}(m1, m2), m)])$, $w_0 = w(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{mobile}(m1, m2), m)])$
$\mathit{ll}_1 = \mathit{ll}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)])$,
$\mathit{rl}_1 = \mathit{rl}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{mobile}(m1, m2)), m)])$,
$l_1 = l(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)])$,
$\mathit{md} = \mathit{md}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)])$,
$d_1 = d(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)])$, $d_2 = d(\mathit{sv}[\mathit{mobile}(\mathit{mobile}(m1, m2), \mathbf{occ}(m))])$
$w_1 = w(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(\mathit{mobile}(m1, m2)), m)])$,
$w_2 = w(\mathit{sv}[\mathit{mobile}(\mathit{mobile}(m1, m2), \mathbf{occ}(m))])$,
$w_{11} = w(\mathit{sv}[\mathit{mobile}(\mathit{mobile}(\mathbf{occ}(m1), m2), m)])$,

$w_{12} = w(sv[mobile(mobile(m1, \mathbf{occ}(m2)), m)])$

the induction assertion
$ll_0 + rl_0 = fixed * (md - d_0 + 1)$

and axioms of the specification
$w_0 = w_1 + w_2, w_1 = w_{11} + w_{12}, d_1 = d_0 + 1$
$ll_1 = l_1 * (w_{12} / w_1), rl_1 = l_1 * (w_{11} / w_1), l_1 = l_0 - fixed,$
$ll_0 = l_0 * (w_2 / w_0), rl_0 = l_0 * (w_1 / w_0)$

it holds:

The induction assertion implies:
$ll_0 + rl_0 = fixed * (md - d_0 + 1)$ iff $l_0 * (w_2 / w_0) + l_0 * (w_1 / w_0) = fixed * (md - d_0 + 1)$
iff $l_0 * ((w_2 + w_1) / w_0) = fixed * (md - d_0 + 1)$ iff $l_0 = fixed * (md - d_0 + 1)$

lhs:
$ll_1 + rl_1 = l_1 * (w_{12} / w_1) + l_1 * (w_{11} / w_1) = l_1 = l_0 - fixed =$
$fixed * (md - d_0 + 1) - fixed = fixed * (md - d_0)$

rhs:
$fixed * (md_1 - d_1 + 1) = fixed * (md - d_0 -1 + 1) = fixed * (md - d_0)$

proof of (6)

analogous to (5). ♦

5.2.4 Semi-Algorithm

In this section we develop a semi-algorithm for the attributed term induction inspired by [Hennicker 92] and the ISAR system [Bauer, Hennicker 93]. This semi-algorithm can be used to calculate complete sets of occurrence terms. It was the basis for the implementation of a system performing attributed term induction [Duschl 94, Weiß 95].

Let *ASpec* = <(*S*, *C*, *F*), F_{Attr}, *Ax*> be an attributed algebraic specification. In order to show, that *ASpec* satisfies an occurrence property *P*:

$$\forall\ c[z_{s_t}] \in (T_{(S,\,C,\,\varnothing)}(\{z_{s_t}\}))_{s_{root}}.\ \forall\ t \in (T_{(S,\,C,\,\varnothing)})_{s_t}.\ P(c[\mathbf{occ}(t)])$$

must be valid. The first quantifier is treated using a special kind of context induction and the second one using term induction. The combination of the induction principles is compatible with the induction ordering defined in section 5.2.3. Thus this induction ordering is used.

As a first step we present the semi-algorithm for the term induction and afterwards the procedures for the context induction are discussed.

$P(c[\mathbf{occ}(t)])$ has to be valid for all ground contexts. But in the presented procedures contexts are constructed containing identifiers. Therefore all ground substitutions have to be applied to the generated contexts to get ground contexts.

The validity of the property $P_{c[z_{s_t}]}$ for a given context $c[z_{s_t}] \in T_{(S,\,C,\,\varnothing)}(\{\ z_{s_t}\ \} \cup X)$ is defined by

$P_{c[z_{s_t}]} = \mathit{true}$, if $\forall\, t \in (T_{(S,\,C,\,\varnothing)})_{s_t}$ and all ground substitutions σ over $(S, C, \varnothing)$
$P(\sigma(c[\mathbf{occ}(t)]))$ is valid.

Starting with a given context $c[z_{s_t}]$ the quantification over all ground terms and all ground substitutions has to be discussed. The proof is done by induction using the syntactical subterm ordering as an ordering. Therefore all smallest terms have to be considered in the base of the induction. I.e. it must be shown that $P(\sigma(c[\mathbf{occ}(f)]))$ is valid for all constants f of sort s_t and all ground substitutions σ with given context $c[z_{s_t}]$. In the term induction step terms must have the form $f(t_1, t_2,..., t_n)$ with f: $s_1, s_2,..., s_n \rightarrow s$ of C and arbitrary terms $t_1, t_2,..., t_n$ of appropriate sort. Thus all constructor symbols with result sort s have to be considered and a nested term induction must be performed for the argument sorts. If the base of the term induction was performed for a sort s_j then the nested term induction can be neglected and a new constant of sort s_j can be introduced which simulates all ground terms of that sort. Otherwise the procedure *nested_term_induction* is invoked which successively performs for all identifiers of the term a term induction.

If a ground occurrence term is reached the property must be valid for the actual occurrence term.

Several induction assertions are valid depending on the case analysis of the context induction. We generalize the induction assertions of the attributed term induction using the presented induction ordering. The notation $IH(t) = \{\, P\sigma \mid \sigma(sv[\mathbf{occ}(x)]) <_{occ} t$ and $\sigma \in Subst \,\}$ is used for the induction assertions of a term t.

procedure *term_induction*

Input:
- $c_1[z_{s_t}]$ actual context

```
begin
(1) for all ($f$: $\rightarrow s_t$) $\in C$ do
(2) if $c_1[f]$ is ground
      then $P(c_1[\mathbf{occ}(f)])$ has to be valid under the assumption that the induction assertions
              $IH(c_1[\mathbf{occ}(f)])$ are valid
      else nested_term_induction($c_1[\mathbf{occ}(f)]$, $\{ s_t \}$, $\varnothing$) fi od
(3) for all ($f$: $s_1, s_2,..., s_n \rightarrow s_t$) $\in C$ do
(4) $t_1 \equiv f(x_1, x_2,..., x_n)$ such that $x_m = \begin{cases} \alpha_m, & \text{if } s_m = s_t \\ x_m, & \text{otherwise} \end{cases}$
      with (new) identifiers $x_1, x_2,..., x_n$ and (new) constants $\alpha_1, \alpha_2,..., \alpha_n$.
(5) $NC = \{ \alpha \mid \alpha$ is new constant in $t_1 \}$
(6) if $c_1[t_1]$ is ground
      then $P(c_1[\mathbf{occ}(t_1)])$ has to be valid under the assumption that the induction assertions
              $IH(c_1[\mathbf{occ}(t_1)])$ are valid
      else nested_term_induction($c_1[\mathbf{occ}(t_1)]$, $\{ s_t \}$, NC) fi od
end
```

In the procedure *nested_term_induction* on the one side all ground substitutions and on the other side all ground terms of the occurrence terms are constructed. This is done fixing an identifier and performing for the sort of the identifier a term induction. It is achieved usingrecursion that step by step all identifiers are substituted by ground terms.

The obtained procedure is nearly the same one as in the term induction case with the exception that the insertion place is not fixed in the term induction procedure:

procedure *nested_term_induction*

Input:
- t_1 actual occurrence term
- S sorts for which the base of the term induction was already performed
- NC set of new constants

```
begin
(1) let $x \in var(t_1)$
(2) for all $(f: \rightarrow sort(x)) \in C$ do
(3) if $t_1[f / x]$ is ground
        then $P(t_1[f / x])$ has to be valid under the assumption that the induction assertions
                $IH(t_1[f / x])$ are valid
        else nested_term_induction($t_1[f / x]$, S, NC) fi od
(4) $S = S \cup \{ sort(x) \}$
(5) for all $(f: s_1, s_2,..., s_n \rightarrow sort(x)) \in C$
(6) $t_2 \equiv f(x_1, x_2,..., x_n)$ with $x_m = \begin{cases} x_m, & \text{if } s_m \notin S \\ \alpha_m, & \text{if } s_m \in S \end{cases}$
        and (new) identifier $x_1, x_2,..., x_n$ and (new) constants $\alpha_1, \alpha_2,..., \alpha_n$.
(7) $NC = NC \cup \{ \alpha \mid \alpha$ is new constant in $t_2 \}$
(8) If $t_1[t_2 / x]$ is ground
        then $P(t_1[t_2 / x])$ has to be valid under the assumption that the induction assertions
                $IH(t_1[t_2 / x])$ are valid
        else nested_term_induction($t_1[t_2 / x]$, S, NC) fi od
end
```

Thus these procedures show the validity of $P_{c[z_s]}$ for a given context $c[z_s]$.

To show the validity of the occurrence property P for all occurrence terms, $P_{c[z_s]}$ must be valid for all contexts $c[z_s]$.

In the base of the context induction $P_{c_1[z_{s_t}]}$ must be valid for all minimal outer contexts $c_1[z_{s_t}]$ of the root sort. If s_t is the root sort then the set of minimal contexts is $\{ z_{s_t} \}$ and the validity of $P_{z_{s_t}}$ has to be proved in the base of the context induction.

Otherwise the set of minimal contexts is defined by

$\{ f(..., x_{i-1}, c[z_{s_t}], x_{i+1},...) \mid (f: s_1, s_2, \ldots, s_n \rightarrow s_{root}) \in C$ and $c[z_{s_t}]$ is an element of the minimal contexts of sort s_i and new identifiers ..., x_{i-1} and x_{i+1},... $\}$

But constructor symbols can exist with an argument sort s_i such that there is no context $c[z_{s_t}]$ of sort s_i. This fact can be shown using signature flow analysis. A recursive call can be neglected for these argument sorts.

All minimal outer contexts of sort s_j have to be constructed for the other argument sorts s_j. These contexts are built using a nested context induction on sort s_j.

The typical dilemma in doing induction proofs consists in finding induction assertions which are general enough to finish the proof successfully. In the proof principle of context induction (cf. [Hennicker 92; Bauer, Hennicker 94]) the problem of finding an appropriate induction assertion comes up choosing a suitable context before doing a nested context induction. Let $c_1[z_{s_i}]$ be the actual context. If the nested context induction can be performed with the context $c_2[z_{s_i}]$ and the implication $P_{c_2[c[z_{s_t}]]} \Rightarrow P_{c_1[c[z_{s_t}]]}$ is valid for all contexts $c[z_{s_t}]$, then the proof obligation for the context $c_1[c[z_{s_t}]]$, namely $P_{c_1[c[z_{s_t}]]}$, is valid. These considerations are implemented in the procedure *context_induction*:

procedure *context_induction*

```
begin
(1) if $s_t = s_{root}$
        then term_induction(occ($z_{s_t}$))
             context_induction_step
(2)     else for all ($f: s_1, s_2,..., s_n \rightarrow s_{root}$) $\in C$ do
(3)          for all $i \in \{ 1, 2,..., n \}$ do
(4)            if $\exists$ context $c[z_{s_t}] \in T_\Sigma(\{ z_{s_t} \})_{s_i}$
(5)              then $c_1[z_{s_i}] \equiv f(x_1, x_2,..., x_n)[z_{s_i} / x_i]$ with (new) identifiers $x_1, x_2,..., x_n$
(6)                   select a $\Sigma$-context $c_2[z_{s_i}]$ such that $P_{c_2[c[z_{s_t}]]}$ is valid if $P_{c_1[c[z_{s_t}]]}$ is valid
                      for all contexts $c[z_{s_t}]$.
(7)                   nested_context_induction($c_2[z_{s_i}]$) fi od od fi
end
```

The proof obligation of the nested context induction is for a given context $c_1[z_{s_i}]$: for all minimal contexts $c[z_{s_t}]$ of sort s_i, $P_{c_1[c[z_{s_t}]]}$ has to be valid. The correctness proof is implemented using a nested context induction on sort s_i. The obtained procedure is nearly the same as for the context induction case with the exceptions that the minimal outer contexts are embedded in the actual context and the sort of the minimal outer contexts is the argument sort of *f*, for which the nested context induction is performed, instead of the root sort:

procedure *nested_context_induction*

Input:

- $c_1[z_s]$ actual context

```
begin
(1) if $s_t = s$
        then term_induction($c_1$[occ($z_s$)])
             context_induction_step
(2)     else for all ($f: s_1, s_2,..., s_n \rightarrow s$) $\in C$ do
(3)          for all $i \in \{ 1, 2,..., n \}$ do
(4)            if $\exists$ context $c[z_{s_t}] \in T_\Sigma(\{ z_{s_t} \})_{s_i}$
(5)              then $c_2[z_{s_i}] \equiv f(x_1, x_2,..., x_n)[z_{s_i} / x_i]$ with (new) identifiers $x_1, x_2,..., x_n$
(6)                 select a $\Sigma$-context $c_3[z_{s_i}]$ such that $P_{c_2[c[z_{s_t}]]}$ is valid if $P_{c_3[c[z_{s_t}]]}$ is valid
                    for all contexts $c[z_{s_t}]$.
(7)                 nested_context_induction($c_1[c_3[z_{s_i}]]$) fi od od fi
end
```

Up to now the context induction step was neglected. $P_{c[z_{s_t}]}$ must be valid under the assumption that $P_{c_1[z_{s_t}]}$ is valid for all outer context $c_1[z_{s_t}]$ of $c[z_{s_t}]$.

Starting with a given context $c_1[z_{s_t}]$ a context $c_1[c_2[z_{s_t}]]$ must be constructed. The set of contexts for $c_2[z_{s_t}]$ is

(*){ $f(\ldots, x_{i\text{-}1}, c[z_{s_t}], x_{i+1}, \ldots)$ | $(f\colon s_1, s_2, \ldots, s_n \rightarrow s_{root}) \in C$ and $c[z_{s_t}]$ is an element of the minimal contexts of sort s_i and fresh identifiers ..., $x_{i\text{-}1}$ and x_{i+1},... }

Again constructor symbols with an argument sort s_i can exist such that there is no context $c[z_{s_t}]$ of sort s_i. Therefore the nested induction can be neglected for these argument sorts.

All minimal contexts of sort s_j have to be considered for the other argument sorts s_j. This is done using a nested context induction on sort s_j. An appropriate context can be selected before the recursive call.

```
procedure context_induction_step
 begin
 for all (f: $s_1, s_2, \ldots, s_n \rightarrow s_t$) $\in$C do
 for all i $\in$ { 1, 2,..., n } do
   if $\exists$ context $c[z_{s_t}] \in T_\Sigma(\{ z_{s_t} \})_{s_i}$
     then $c_2[z_{s_i}] \equiv c_1[f(x_1, x_2, \ldots, x_n)[z_{s_i} / x_i]]$ for an arbitrary context $c_1$ of the root sort and
          (new) identifiers $x_1, x_2, \ldots, x_n$.
          select a $\Sigma$-context $c_3[z_{s_i}]$ such that $P_{c_2[c[z_{s_t}]]}$ is valid if $P_{c_3[c[z_{s_t}]]}$ is valid for all
          contexts $c[z_{s_t}]$.
          nested_context_induction_step($c_3[z_{s_i}]$) fi od od
 end
```

In the nested context induction we start with an actual context $c[z_{s_i}]$ of the form $c_1[c_2[z_{s_i}]]$ with the context $c_1[z_{s_i}]$ of depth n and some context $c_2[z_{s_i}]$. If s_i is the sort s_t, $P_{c_1[c_2[z_{s_t}]]}$ must be valid, since $c_2[z_{s_i}]$ is a minimal context. Otherwise, i.e. s_i is not the sort s_t, the proof obligation is $P_{c_1[c_2[c_3[z_{s_t}]]]}$ for some context $c_3[z_{s_t}]$ with minimal context $c_2[c_3[z_{s_t}]]$. The contexts $c_3[z_{s_t}]$ are the set (*) from above.

Again a constructor symbol with an argument sort s_i can exist such that there is no context $c[z_{s_t}]$ of sort s_i. Therefore the nested induction can be neglected for these argument sorts.

All minimal contexts of sort s_j have to be considered for the other argument sorts s_j. This is done using a nested context induction on sort s_j. An appropriate context can be selected before a nested induction is done.

procedure *nested_context_induction_step*

Input:
- $c_1[z_s]$ actual context

```
begin
(1) if $s_t = s$
      then term_induction($c_1$[occ($z_s$)])
(2)   else for all ($f: s_1, s_2,..., s_n \rightarrow s) \in C$ do
(3)          for all $i \in \{ 1, 2,..., n \}$ do
(4)            if $\exists$ context $c[z_{s_t}] \in T_\Sigma(\{ z_{s_t} \})_{s_i}$
(5)              then $c_2[z_{s_i}] \equiv f(x_1, x_2,..., x_n)[z_{s_i} / x_i]$ with (new) identifiers $x_1, x_2,..., x_n$
(6)                   select a $\Sigma$-context $c_3[z_{s_i}]$ such that $P_{c_2[c[z_{s_t}]]}$ is valid if $P_{c_3[c[z_{s_t}]]}$ is valid for all
                      contexts $c[z_{s_t}]$.
(7)                   nested_context_induction_step($c_1[c_3[z_{s_i}]]$) fi od od fi
end
```

Thus the whole semi-algorithm for the attributed term induction is developed. In order to show that an occurrence property *P* is valid for an attributed algebraic specification *ASpec*, the procedure *context_induction* has to be called. Note, that the semi-algorithm calculates a complete set of occurrence terms.

5.2.5 Heuristics

Efficient heuristics for the semi-algorithm and the attributed term induction can be obtained using attributed signature flow analysis.

Attributed signature flow analysis can be used

- to optimize and
- to generalize

the proofs.

The application of signature flow analysis can be optimizable and necessary.

The use of signature flow analysis is necessary either if

- an infinite proof would be obtained, e.g. an insertion place is tried to be constructed but in this subterm no such insertion place exists, or
- the obtained induction assertion are too restrictive.

In the first case the system can realize that no insertion place exists in the subterm or that the the construction of the subterm is infinite. In the case of infiniteness the information from the analysis technique can be used to generalize the context to get a finite proof.

The obtained informations and the generalizations can also be necessary, if the induction assertions are too weak, since generalized contexts deliver stronger induction assertions to finish the proof successfully.

The technique can be helpful and optimizable in cases where the search space can be cut knowing the signature flow analysis information, e.g. no insertion place exists in this subterm or the context can be generalized such that a context is obtained, for which the proof was already performed.

Other heuristics of the proof principle can take into consideration the knowledge of the attribute dependencies. E.g. the knowledge, that the attributes of the property to prove are only passed through and not changed in a subtree, implies that the subterms can be neglected for the proof.

5.3 Existentially Quantified Formulae

Three pure kinds of existentially quantified formulae can be found in the framework of attributed algebraic specifications, e.g.

- $\exists\, x_{Nat}.\ succ(x) = succ(succ(zero))$, corresponding to usual narrowing,
- $\exists\, x_s.\ f_{Attr}(t) = x$ with attribute function symbol f_{Attr} and occurrence term t of appropriate sort, corresponding to attribute value calculation,
- $\exists\, sv_{s \rightarrow s'}.\ \exists\, x_s.\ f_{Attr}(sv[\mathbf{occ}(x)]) = t$ with attribute function symbol f_{Attr} and term t, corresponding to the construction of a syntax tree.

For all these cases the most general solutions has to be calculated. The notion of solution is defined as:

Definition 5.3.1 (solution)

Let *ASpec* = <Σ, F_{Attr}, *Ax*> be an attributed algebraic specification. A substitution σ is called *solution for a set of equations E wrt. Ax* if

$$Mod(ASpec) \models_{\text{attr}} (t = r)\sigma \text{ for all } (t = r) \in E$$

A solution σ is *more general* than a solution τ iff $\sigma \leq \tau$ ♦

Given an attributed algebraic specification and a fixed term to be attributed, the instantiated attribute equations of this term have to be considered. In these equations the attribute occurrences are like identifiers for which a solution has to be computed, similar to [Chirica, Martin 79].

Therefore the ∃-closure of an equation, which has to be solved, is considered introducing new identifiers for the attribute occurrences and new equations specifying the equality between a new identifier and the corresponding attribute occurrence.

Definition 5.3.2 (∃-closure)

Let *ASpec* = <Σ, F_{Attr}, *Ax*> be an attributed algebraic specification and $t = r$ an equation with $t, r \in AT_\Sigma(SV)_s$ for some sort $s \in sorts(\Sigma)$. The *∃-closure* of $t = r$ is defined as

$$(*)\ \exists\text{-}closure(t = r) = (t' = r' \wedge x_1 = a_1 \wedge x_2 = a_2 \wedge \ldots \wedge x_n = a_n)$$

if the attribute occurrences in t and r are $a_1, a_2, \ldots, a_n$; $x_1, x_2, \ldots, x_n$ are new identifiers and t' and r' are obtained from t and r replacing the attribute occurrences by the corresponding new identifiers.

Having a set of equations E of the form $\{\, x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n \,\}$ for identifiers $x_1, x_2, \ldots, x_n$ and attribute occurrences $a_1, a_2, \ldots, a_n$ the ∃-closure is defined as

$$(**)\ \exists\text{-}closure(E, t = r) = (t' = r' \wedge x_{i_1} = a_{i_1} \wedge \ldots \wedge x_{i_k} = a_{i_k} \wedge y_1 = b_1 \wedge \ldots \wedge y_l = b_l)$$

if the attribute occurrences in t and r are $a_{i_1}, a_{i_2}, \ldots, a_{i_k}, b_1, b_2, \ldots, b_l$ with $a_{i_1}, a_{i_2}, \ldots, a_{i_k} \in \{\, a_1, a_2, \ldots, a_n \,\}$; $y_1, y_2, \ldots, y_l$ are new identifiers and t' and r' are obtained from t and r replacing the attribute occurrences by the corresponding identifiers.

The following projection functions are defined:

$$AttrEq(\exists\text{-}closure(t = r)) = (t' = r'),$$

$IdEq(\exists\text{-}closure(t = r)) = \{\ x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n\ \}$

if the ∃-*closure* has the form (*) and

$AttrEq(\exists\text{-}closure(E, t = r)) = (t' = r'),$

$IdEq(\exists\text{-}closure(E, t = r)) = \{\ x_{i_1} = a_{i_1}, \ldots, x_{i_k} = a_{i_k}, y_1 = b_1, \ldots, y_l = b_l\ \}$

if the ∃-*closure* has the form (**). ◆

The solutions obtained for the ∃-*closure* of an equation (restricted to the identifiers of the original equation) are the same solutions as for the equation itself:

Lemma 5.3.3

Let *ASpec* be an attributed algebraic specification, *E* be a set of equations over *ASpec* and σ be a solution for *E*, then it holds

$Mod(ASpec) \models_{\text{attr}} (t = r)\sigma$ for all $(t = r) \in E$ iff
$Mod(ASpec) \models_{\text{attr}} \exists\text{-}closure(t = r)\sigma'$ for all $(t = r) \in E$ with σ is the restriction of σ' to the identifiers of *t* and *r*. ◆

Proof

obvious ◆

5.3.1 Solving Equations with Subterm Identifiers

In this section a motivation and application areas for solving equations with subterm identifiers are given. Afterwards we present a small example showing the idea how such equations can be solved. These considerations lead to the *attributed narrowing calculus*.

5.3.1.1 Motivation and Application Areas

Pure attribute grammars are called *static* since the attribution has no influence on the analysis of the syntax and building the abstract syntax tree. Therefore in [Ganzinger 78] dynamic attribute grammars are introduced. The main difference to pure attribute grammars consists in assigning conditions over the attribute values of a non terminal to the various productions of that non terminal. Abstract syntax trees satisfying the conditions may be constructed, i.e. partial attribute grammars are obtained.

We extend the notion of dynamic attribute grammars in the following way: Firstly, we abstract from the analysis of the syntax, i.e. parsing. The conditions are used for deduction to derive programs. Secondly, in [Ganzinger 78] the applicability of a production rule is determined by so-called comparison attributes which are inherited attributes (!). Depending on their values a production rule is selected. In the new approach a term is completed such that a given attribution is satisfied.

The main advantage using attributed algebraic specifications instead of algebraic specifications in this area is the clear distinction between the syntax (to be constructed) and the attribution, describing the semantics of it.

Parsers use error recovery strategies to obtain a syntactical correct program. The new approach completes a syntax tree to obtain a program with correct (statical) semantics.

Application areas of the new technique are e.g.:

- intelligent user guidance: generation of the dialogues to be performed,
- error-recovery: compilers adapting the syntax tree and programs with user interfaces reacting to wrong inputs,
- automatic programming: automatic declaration of identifiers, automatic construction of parts of a program,
- testing on a high level of abstraction.

5.3.1.2 Small Example

In this section a small example is presented demonstrating the proceeding for solving existentially quantified formulae.

The example shows that an infinite set of solutions can be obtained. But the most general solution can be determined such that all other solutions are instances of the determined solution.

The existentially quantified formula to be solved is (with the usual abbreviations):

$$\exists\, sv1_{Mobile \rightarrow Mobile}.\ \exists\, sv2_{Mobile,\, Mobile \rightarrow Mobile}.\ \exists\, n_{Nat}.$$
$$weight(sv1[\mathbf{occ}_{Mobile}(sv2[mobile(cube(n), cube(3)), cube(2)])]) = 6 \qquad \text{(i)}$$

and

$$weight(sv1[sv2[\mathbf{occ}_{Mobile}(mobile(cube(n), cube(3))), cube(2)]]) = 4 \qquad \text{(ii)}$$

and

$$depth(sv1[\mathbf{occ}_{Mobile}(sv2[mobile(cube(n), cube(3)), cube(2)])]) + 1 =$$
$$depth(sv1[sv2[\mathbf{occ}_{Mobile}(mobile(cube(n), cube(3))), cube(2)]]) \qquad \text{(iii)}$$

The obtained solution should be:

$$[\ sv1_{Mobile \rightarrow Mobile} \,/\, sv3_{Mobile \rightarrow Mobile},\ sv2_{Mobile,\, Mobile \rightarrow Mobile} \,/\, mobile(z_{Mobile}, z_{Mobile}),\ n \,/\, 1\ ]$$

$sv1_{Mobile \rightarrow Mobile} / sv3_{Mobile \rightarrow Mobile}$ states that an infinite number of solutions is obtained and only the most general solution is calculated. How can such a formulae be solved? A simple strategy enumerates all possible syntax trees without further information and computes the attribute values. Then it recognizes that the attribution is different to the given one and will adapt the syntax tree. This will be done until an attributed syntax tree is constructed satisfying the property. Using this method all possible syntax trees are enumerated leading to an explosion of the number of considered syntax trees. Furthermore, it is possible, e.g. if variables of a programming language are defined by regular expressions (algebraic specifications are extended by regular expressions e.g. in [Heering, Klint 89]) that an infinite number of variables satisfying the regular expression must be tested or in the mobile example all natural numbers must be enumerated to get the solution.

Therefore a goal-directed change of the syntax tree has to be performed to keep the numbers of syntax trees small and to handle „infinite" domains.

Considering condition (iii) of the formula and axiom (6) of the specification the first solution is obtained:

$$sv2_{Mobile,\, Mobile \rightarrow Mobile} \,/\, mobile(z_{Mobile}, z_{Mobile})$$

resulting in the new formula with solved condition (iii):

$$\exists\, sv1_{Mobile \to Mobile}.\ \exists\, n_{Nat}.$$

$$weight(sv1[\mathbf{occ}_{Mobile}(mobile(mobile(cube(n), cube(3)), cube(2)))]) = 6 \qquad \text{(i)}$$

and

$$weight(sv1[mobile(\mathbf{occ}_{Mobile}(mobile(cube(n), cube(3))), cube(2))]) = 4 \qquad \text{(ii)}$$

Applying axiom (2) and twice axiom (1) to (ii) produces

$$n + 3 = 4$$

It can be derived with the axioms of the natural numbers that n has to be equal one, i.e. the next obtained solution is:

$$[\, n / 1 \,]$$

Condition (i) is satisfied trivially with the calculated solutions. Thus any other subterm identifier of the same functionality can be used for the subterm identifier $sv1$, delivering the above substitution as a solution.

Note, that in the example only directed attribute equations are used for the calculation of the attribute values. Usually undirected attribute equations have to be applied to determine the solutions. Therefore a more complex deduction engine for the attribute evaluation is necessary.

5.3.2 Attributed Narrowing

For a given term the instantiated attribute equations define an equational system which has to be solved, i.e. the attribute occurrences can be viewed as identifiers. We assume in our considerations, that the attribution is acyclic and moreover the axioms of the algebraic core can be used as rewrite rules. The considered rewriting is not a rewriting modulo a set of universally quantified equations, like rewriting modulo commutativity, but modulo a set of existentially quantified formuale, i.e. additional equations have to be solved. For the theory of narrowing see e.g. [Hofbauer, Kutsche 89]. The presented calculus is an extension of the narrowing calculus presented there.

5.3.2.1 Attributed Narrowing Calculus

The starting point for the narrowing calculus is a set of rewrite rules R being the algebraic core of the axioms, i.e. $R = RRules(AlgAx(Ax))$[13], and a set of equations AEq being the attribute equations, i.e. $AEq = AttrAx(Ax)$.

Definition 5.3.2.1.1 (attributed narrowing calculus)

Let R be a set of rewrite rules and AEq be a set of attribute equations. The *attributed narrowing calculus*, denoted by $\vdash_{\text{narrow}}$ consists of the following rules:

$$(N1)\frac{(E \cup \{(t_1 = t_2)[u]\}, \tau, Attr)}{(E\sigma' \cup \{((t_1 = t_2)[r])\sigma'\}, \tau\sigma, Attr)}\ {}^{14}$$

[13] $RRules(\{\, t_1 = r_1, t_2 = r_2, \ldots, t_n = r_n \,\}) = \{\, t_1 \to r_1, t_2 \to r_2, \ldots, t_n \to r_n \,\}$

[14] $(t_1 = t_2)[u]$ denotes that $(t_1 = t_2)$ is either of the form $(c[u] = t_2)$ or $(t_1 = c[u])$ for some context c.

if $(l \rightarrow r) \in R$ and $\sigma' \in mguSet(l, u)$ and
σ is the restriction of σ' to the identifiers of $(t_1 = t_2)$

$$(N2)\frac{(E \cup \{t_1 = t_2\}, \tau, Attr)}{(E\sigma, \tau\sigma, Attr)}$$

if $\sigma \in mguSet(t_1, t_2)$

$$(N3)\frac{(E \cup \{(t_1 = t_2)[x]\}, \tau, Attr \cup \{x = f_{Attr}(t)\})}{(E\sigma' \cup \{((t_1 = t_2)[x])\sigma'\} \cup E', \tau\sigma, Attr\sigma' \cup \{x = f_{Attr}(t)\}\sigma' \cup Attr')}$$

if $\exists\ (r_1 = r_2)[u] \in AEq$, $\sigma' = mguSet(u, f_{Attr}(t))$,
$E' = AttrEq(\exists\text{-}closure(Attr, ((r_1 = r_2)[u])\sigma')$,
$Attr' = IdEq(\exists\text{-}closure(Attr, ((r_1 = r_2)[u])\sigma')$ and
σ is the restriction of σ' to the identifiers of $x = f_{Attr}(t)$ ♦

Note, the differences to usual narrowing:

(N1) and (N2) are usual narrowing rules with the difference that instead of the most general unifier a minimal unifier out of the complete set of minimal unifiers is used.

(N3) is the new one, since equations have to be solved „modulo“ a set of other solvable equations.

Definition 5.3.2.1.2 (attributed narrowing)

Let R be a set of rewrite rules, AEq be a set of attribute equations and E be a set which has to be solved. *Attributed narrowing* is defined as follows:

(1) Start with a triple $(AttrEq(\exists\text{-}closure(E)), [], IdEq(\exists\text{-}closure(E)))$, i.e. with the set of equations to be solved, in which the attribute occurrences are changed to identifiers, the identical substitution and the mapping identifier to attribute occurrence.
(2) If a triple $(\varnothing, \tau, Attr)$ is reached, the restriction of τ to the identifiers of E is called *answer substitution.* ♦

Note, the difference to usual narrowing where it is started with the set E instead of the $\exists$-*closure* of E. The attributed narrowing calculus is sound and complete as the following theorems state:

Theorem 5.3.2.1.3 (soundness of attributed narrowing)

Let R be a set of rewrite rules, AEq be a set of attribute equations and E be a set which has to be solved. Each answer substitution calculated with attributed narrowing is a solution of E wrt. R and AEq. ♦

Proof

The given proof is similar to the one given in [Hofbauer, Kutsche 89] for narrowing.

Let $Ax = Equs(R) \cup AEq$[15].

[15] $Equs(\{ t_1 \rightarrow r_1, t_2 \rightarrow r_2, \ldots, t_n \rightarrow r_n \}) = \{ t_1 = r_1, t_2 = r_2, \ldots, t_n = r_n \}$

It is shown by induction on the length l of the derivation that

$(E, \tau, Attr) \vdash_{\text{narrow}} (\varnothing, \tau\rho, Attr')$

implies that ρ is a solution of E wrt. R and AEq.
$l = 0$:

$l = 0$ implies $(E, \tau, Attr) = (\varnothing, \tau\rho, Attr')$, i.e. $E = \varnothing, \tau = \tau\rho, Attr = Attr'$
But every substitution solves the empty set of equations, especially $\rho = []$.

$l \to l + 1$:

$(N1)$: $(E \cup \{ (t_1 = t_2)[u] \}, \tau, Attr) \vdash_{(N1)} (E\sigma' \cup \{ ((t_1 = t_2)[r])\sigma' \}, \tau\sigma, Attr)$
$\vdash_{\text{narrow}} (\varnothing, \tau\sigma\rho, Attr')$

The induction assertion states:

$Ax \models_{\text{attr}} (E\sigma' \cup \{ ((t_1 = t_2)[r])\sigma' \}\rho$

Since σ is the restriction of σ' to the identifiers of $(t_1 = t_2)$ and $\sigma \in mguSet(l, u)$ it is valid:

$E\sigma' = E\sigma$

Therefore for all $t_1' = t_2' \in E$ holds: $Ax \models_{\text{attr}} t_1'\sigma\rho = t_2'\sigma\rho$ and therefore $\sigma\rho$ is solution of E wrt. R and AEq.
It remains to show $Ax \models_{\text{attr}} t_1\sigma\rho = t_2\sigma\rho$ under the assumption that $Ax \models_{\text{attr}} ((t_1 = t_2)[r])\sigma'\rho$ is valid.

Wlog. it is assumed that $(t_1 = t_2)[u] \equiv c[u] = t_2$, i.e. the rewrite rule is applied in t_1, i.e.
$Ax \models_{\text{attr}} (c[r] = t_2)\sigma'\rho$

Especially:
$Ax \models_{\text{attr}} (c\sigma[r\sigma'] = t_2\sigma)\rho$

is valid, since σ is the restriction of σ' to $var(t_1 = t_2)$.

With the equational calculus $\vdash_{\text{aeq}}$ can be derived:

$Ax \vdash_{(ax)} l = r \vdash_{(\text{subst})} l\sigma' = r\sigma' \vdash_{(fun)}{}^{*} c\sigma[l\sigma'] = c\sigma[r\sigma'] \vdash_{(subst)} (c\sigma[l\sigma'] = c\sigma[r\sigma'])\rho$

The completeness of the equational calculus implies:

$Ax \vdash_{\text{aeq}} (c\sigma[r\sigma'] = t_2\sigma)\rho$

With *(trans)* follows:

$Ax \vdash_{\text{aeq}} (c\sigma[l\sigma'] = t_2\sigma)\rho$

The correctness of the equational calculus and the fact that σ' is in the complete set of minimal unifiers for u and l implies:

$Ax \models_{\text{attr}} t_1\sigma\rho = t_2\sigma\rho$

Thus $\sigma\rho$ is solution of E wrt. R and AEq.

$(N2)$: $(E \cup \{ t_1 = t_2 \}, \tau, Attr) \vdash_{(N2)} (E\sigma, \tau\sigma, Attr) \vdash_{\text{narrow}} (\varnothing, \tau\sigma\rho, Attr')$

The induction assertion states:

$$Ax \models_{\text{attr}} E\sigma\rho$$

Thus $\sigma\rho$ is solution of E. Since $\sigma \in mguSet(t_1, t_2)$

$$Ax \models_{\text{attr}} t_1\sigma = t_2\sigma \text{ implying } Ax \models_{\text{attr}} t_1\sigma\rho = t_2\sigma\rho$$

is valid and therefore

$$Ax \models_{\text{attr}} (E \cup \{\, t_1 = t_2 \,\})\sigma\rho$$

Consequently $\sigma\rho$ is solution for $E \cup \{\, t_1 = t_2 \,\}$.

$(N3)$: $(E \cup \{\, (t_1 = t_2)[x] \,\}, \tau, Attr \cup \{\, x = f_{Attr}(t) \,\}) \vdash_{(N3)}$

$$(E\sigma' \cup \{\, ((t_1 = t_2)[x])\sigma' \,\} \cup E', \tau\sigma, Attr\sigma' \cup \{\, x = f_{Attr}(t)\}\sigma' \cup Attr')$$
$$\vdash_{\text{narrow}} (\varnothing, \tau\sigma\rho, Attr'')$$

The induction assertion states:

$$Ax \models_{\text{attr}} (E\sigma' \cup \{\, ((t_1 = t_2)[x])\sigma' \,\} \cup E')\rho$$

Since σ is the restriction of σ' to the identifiers of $f_{Attr}(t)$ (x is a new identifier) and $f_{Attr}(t)$ is not in E, $E\sigma' \equiv E\sigma$ and $(t_1 = t_2)[x])\sigma \equiv (t_1 = t_2)[x])\sigma'$ is valid, especially:

$$Ax \models_{\text{attr}} (E\sigma \cup \{\, ((t_1 = t_2)[x])\sigma \,\}\rho$$

Consequently $\sigma\rho$ is a solution for $E \cup \{\, (t_1 = t_2)[x] \,\}$. ♦

Theorem 5.3.2.1.4 (completeness of the attributed narrowing calculus)

Let R be a confluent rewriting system, AEq be an acyclic attribution and E be a set of equations, then for each normalized substitution σ for E wrt. R and AEq a substitution τ can be derived from $(AttrEq(\exists\text{-closure}(E)), [], IdEq(\exists\text{-closure}(E)))$ with $\tau \le \sigma$.

(A substitution σ is called *normalized*, if $sv\sigma$ is in normal form wrt. R for all identifiers $sv \in SV$, i.e. cannot be reduced.)

Proof

Let $Ax = Equs(R) \cup AEq$. By assumption $Ax \models_{\text{attr}} t_1\sigma = t_2\sigma$,
iff $Ax \models_{\text{attr}} t_1\sigma = t_2\sigma \wedge AttrAx(Ax)$
especially $Ax \models_{\text{attr}} t_1\sigma = t_2\sigma \wedge AttrAx(Ax)\sigma$
iff $Ax \models_{\text{attr}} \exists\text{-}closure(t = r)$ for all $t = r \in \{\, t_1\sigma = t_2\sigma \,\} \cup AttrAx(Ax)\sigma$

Since R is a set of confluent rewrite rules it holds:

$$\begin{tikzcd} t\sigma \arrow[dr] & & r\sigma \arrow[dl] \\ & t' & \end{tikzcd}$$

and because rewriting is a special case of $(N1)$, it holds:

$$(\{\, t\sigma = r\sigma \,\}, [], Attr) \vdash_{(N1)}{}^{*} (\{\, t' = t' \,\}, [], Attr') \vdash_{(N2)} (\varnothing, [], Attr'')$$

This derivation is valid for all equations in the $\exists\text{-}closure(t = r)$ for all $t = r \in \{\, t_1\sigma = t_2\sigma \,\} \cup AttrAx(Ax)\sigma$, since $\exists$-closure contains only usual terms which are rewritten.

With $(N3)$ each $(t = r) \in AttrAx(Ax)$ can be derived, i.e.

$$(E, [], Attr) \vdash_{(N3)} (E \cup \{ t = r \}, [], Attr'\sigma) \vdash_{\text{narrow}} (\varnothing, [], Attr')$$

It remains to show that

$$(E\sigma, [], Attr) \vdash_{\text{narrow}} (\varnothing, [], Attr') \text{ implies } (E, [], Attr) \vdash_{\text{narrow}} (\varnothing, \tau, Attr') \text{ with } \tau \leq \sigma.$$

This can be shown by induction on the length of the derivation as in the narrowing calculus. ♦

Examples using the attributed narrowing calculus are given in the user interface (section 8.1.4) and the compiler (section 8.2.4) case study.

Remark

If a total ordering exists on the attribute occurrences of an attribute equation of the form

$$f_{Attr}(t) = c[f_{Attr_1}(t_1), f_{Attr_2}(t_2), \ldots, f_{Attr_n}(t_n)]$$

such that $f_{Attr_1}(t_1), f_{Attr_2}(t_2), \ldots, f_{Attr_n}(t_n)$ are the only attribute occurrences on the right hand side of the equation and $f_{Attr_1}(t_1) < f_{Attr}(t), f_{Attr_2}(t_2) < f_{Attr}(t), \ldots, f_{Attr_n}(t_n) < f_{Attr}(t)$ is the total ordering, then this attribute equation can be written as a rewrite rule:

$$f_{Attr}(t) \rightarrow c[f_{Attr_1}(t_1), f_{Attr_2}(t_2), \ldots, f_{Attr_n}(t_n)]$$

Such a rewrite rule can be added to the rewrite rules of the algebraic core.

5.3.2.2 Attribute Value Calculation

The attribute value calculation is a special case of solving existentially quantified formulae using attributed narrowing. The attribute evaluation ordering is discussed elsewhere.

Definition 5.3.2.2.1 (attribute value calculation)

The attribute value calculation for a constructor term t is defined by:

(1) The start configuration for the attributed narrowing calculus is $(E, [], \varnothing)$ with the set

$$E = \bigcup_{t^{occ} \in OccTerm(t)} \left\{ \exists x^{t^{occ}} . f(t^{occ}) = x^{t^{occ}} \right\} \text{ and the empty substitution } [].$$

(2) Apply the rules $(NR1)$ to $(NR3)$ until a pair $(\varnothing, \sigma, Attr)$ is reached being the empty set of equations, the answer substitution σ, i.e. the attribute values, and an auxiliary set $Attr$.

Theorem 5.3.2.2.2 (soundness of the attribute value calculation)

The attribute value calculation is sound, i.e. it holds $Mod(ASpec) \models_{\text{attr}} E\sigma$. ♦

Proof

Follows from the soundness of the attributed narrowing calculus. ♦

6 Refinements

Using attributed algebraic specifications in the software engineering process the notion of implementation relation has to be formalized and a proof theoretical characterization has to be investigated. The starting point in the software engineering process is an abstract attributed algebraic specification $ASpec_1$. $ASpec_1$ can be implemented by a specification $ASpec_2$ which in turn is refined until a functional attribute grammar $ASpec_n$ is reached after several implementation steps from which an efficient program can be generated (see figure 20):

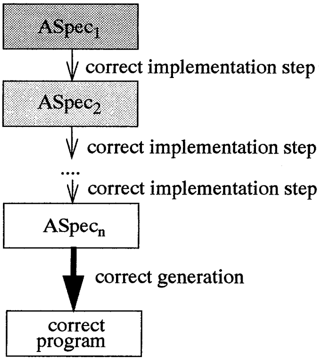


figure 20: software engineering process with attributed algebraic specifications

The result of this software engineering process is a correct program if the implementation relation is transitive, the correctness of each implementation step is shown and the generation process is correct.

This software engineering process is comparable to a process in which it is started with a loose algebraic specification and it is reached an algebraic specification in a functional way after several implementation steps from which a program can be generated. But an attribute grammar is more abstract than a functional program.

Several design decisions are made proceeding from a specification to an attribute grammar. These include decisions how to perform the attribution (attributes and attribute dependencies, but not phases), concerning the concrete representation of the abstract data types of the included algebraic specification, or choice of algorithms which are left open using high-level specifications.

6.1 Aims

The aims of the implementation relations for attributed algebraic specifications are:

- The attribution of a specification may be changed, especially new attributes may be introduced, a several pass attribution may be replaced by a one pass attribution and vice versa and the attribute dependencies may be modified.
- The implementation relation has to be transitive to use the implementation relation in the described software engineering process.
- The intuitive notion of behaviour has to be supported by a behavioural implementation

relation.

- Structuring mechanisms have been defined for attributed algebraic specifications to handle complex specifications. The implementation relations have to be monotone wrt. the specification building operations to benefit from these mechanisms in the software engineering process, i.e. let f be a specification building operation and $ASpec_1$ be an implementation of $ASpec_2$ then $f(ASpec_1)$ has to be an implementation of $f(ASpec_2)$.
- A proof theoretical characterization, which can be handled by a system, has to be given to perform implementation proofs (semi-)automatically.
- The starting point may be a loose specification.

In the following two kinds of implementation relations are defined, a standard implementation relation, which does not concern the behavioural aspects, and a behavioural implementation relation, taking the observability issues into consideration.

6.2 Standard Implementations

The standard implementation relation for the new approach is defined analogous to the refinement or model class inclusion known from algebraic specifications (cf. e.g. [Wirsing 90]), i.e. a specification $ASpec_1$ is implemented by a specification $ASpec_2$ if the model class of $ASpec_2$ restricted to the signature of $ASpec_1$ reachable by the constructors of $ASpec_1$ is a subset of the model class of $ASpec_1$ and the signature of $ASpec_2$ may be an extension of $ASpec_1$.

Definition 6.2.1 (standard implementation)

An attributed algebraic specification $ASpec_2$ is a *standard implementation* of an attributed algebraic specification $ASpec_1$ (written $ASpec_1 \rightsquigarrow ASpec_2$), iff

$sig(ASpec_1) \subseteq sig(ASpec_2)$ and for all models $B \in Mod(ASpec_2)$ holds
$<B|_{sig(ASpec_1)}> \in Mod(ASpec_1)$. ♦

$ASpec_2$ is a standard implementation of an attributed algebraic specification $ASpec_1$ if the axioms of $ASpec_1$ are valid in $ASpec_2$.

Theorem 6.2.2 (proof theoretical characterization)

Let $ASpec_1 = <\Sigma_1, F_{Attr1}, Ax_1>$ and $ASpec_2 = <\Sigma_2, F_{Attr2}, Ax_2>$ be attributed algebraic specifications. To prove that $ASpec_1$ is implemented by $ASpec_2$ it is sufficient to show

$Ax_2 \models_{attr} \sigma(Ax_1)$ for all ground substitutions σ over $sig(ASpec_1)$

under the assumption that $sig(ASpec_1) \subseteq sig(ASpec_2)$. ♦

Proof

Let $Ax_2 \models_{attr} \sigma(Ax_1)$ for all ground substitutions σ over $sig(ASpec_1)$,

i.e. $Mod(ASpec_2) \models_{attr} \sigma(Ax_1)$ for all ground substitutions σ over $sig(ASpec_1)$,

i.e. $\forall A \in Mod(ASpec_2)$ and for all valuation v holds:

$I_v^A[\sigma(t)] = I_v^A[\sigma(r)]$ for all $t = r \in Ax_1$ and for all ground substitutions σ over $sig(ASpec_1)$.
Let $A \in Mod(ASpec_2)$ be an arbitrary model and v an arbitrary valuation.

Since t and r are terms over the signature of $ASpec_1$ holds:

$I_v^{A|\Sigma_1}[\sigma(t)] = I_v^{A|\Sigma_1}[\sigma(r)]$ for all $t = r \in Ax_1$ and for all ground substitutions σ over $sig(ASpec_1)$, i.e.

$A|_{\Sigma_1} \models_{\text{attr}} \sigma(t) = \sigma(r)$ for all $t = r \in Ax_1$ and for all ground substitutions σ over $sig(ASpec_1)$.

In order to prove that $A|_{\Sigma_1}$ is in the model class of $ASpec_1$ the algebra $A|_{\Sigma_1}$ has to be reachable with $cons(\Sigma_1)$. But $A|_{\Sigma_1} \models_{\text{attr}} \sigma(t) = \sigma(r)$ for all $t = r \in Ax_1$ and for all ground substitutions σ over $sig(ASpec_1)$ implies $<A|_{\Sigma_1}> \models_{\text{attr}} \sigma(t) = \sigma(r)$ for all $t = r \in Ax_1$ and for all ground substitutions σ over $sig(ASpec_1)$. ♦

Using the attributed equational calculus a proof theoretical characterization of the standard implementation relation is obtained:

Corollary 6.2.3

Let $ASpec_1 = <\Sigma_1, F_{Attr_1}, Ax_1>$ and $ASpec_2 = <\Sigma_2, F_{Attr_2}, Ax_2>$ be attributed algebraic specifications. To prove that $ASpec_1$ is implemented by $ASpec_2$ it is sufficient to show

$Ax_2 \vdash_{\text{aeq}} \sigma(Ax_1)$ for all ground substitutions σ over $sig(ASpec_1)$

under the assumption that $sig(ASpec_1) \subseteq sig(ASpec_2)$. ♦

Proof

Follows immediately from the soundness and completeness of the equational calculus and Theorem 6.2.2. ♦

Example 6.2.4

An example for a trivial implementation step is the refinement of *LMOBILE* by *CMOBILE* since only the axioms (10)-(13) are added to *LMOBILE*.

Fact 6.2.5

It holds:

$LMOBILE \rightsquigarrow CMOBILE$ ♦

Proof

trivial, since only the axioms (10)-(13) are added to *LMOBILE*. ♦

More interesting implementation steps are obtained considering the relation of *CMOBILE* and *CMOBILE*1 and of *LMOBILE* and *CMOBILE*1, respectively.

Fact 6.2.6

It holds:

$CMOBILE \rightsquigarrow CMOBILE1$ ♦

Proof

The axioms of *CMOBILE* must be derived with the axioms of *CMOBILE*1. The only axioms of *CMOBILE* not belonging to the axioms of *CMOBILE*1 are

(1) $\mathit{length}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) =$
$\mathit{leftlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) + \mathit{rightlenght}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))])$

(2) $\mathit{weight}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(m1, m2))]) * \mathit{leftlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) =$
$\mathit{weight}(\mathit{sv}[\mathit{mobile}(m1, \mathbf{occ}(m2))]) * \mathit{rightlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))])$

(3) $\mathit{length}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) =$
$\mathit{fixed} * (\mathit{cmaxdepth}(\mathbf{occ}(\mathit{sv}[\mathit{mobile}(m1, m2)])) - \mathit{depth}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]))$

(1) and (2) are left to the reader. (3) was shown in section 5.2.

Fact 6.2.7

It holds:

$\mathit{LMOBILE} \leadsto \mathit{CMOBILE}1$ ♦

Proof

The axioms of *LMOBILE* must be derived with the axioms of *CMOBILE*1. The only axioms of *LMOBILE* not belonging to the axioms of *CMOBILE*1 are

(1) $\mathit{length}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) =$
$\mathit{leftlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) + \mathit{rightlenght}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))])$

(2) $\mathit{weight}(\mathit{sv}[\mathit{mobile}(\mathbf{occ}(m1, m2))]) * \mathit{leftlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))]) =$
$\mathit{weight}(\mathit{sv}[\mathit{mobile}(m1, \mathbf{occ}(m2))]) * \mathit{rightlength}(\mathit{sv}[\mathbf{occ}(\mathit{mobile}(m1, m2))])$

The proofs of (1) and (2) are the same as in the implementation proof of *CMOBILE* $\leadsto$ *CMOBILE*1, since in specification *CMOBILE* only the axioms (10)-(13) are added to *LMOBILE* which are not used in the proof. ♦

But *BEHLMOBILE* cannot be implemented by *CMOBILE*2

Fact 6.2.8

It does not hold:

$\mathit{BEHLMOBILE} \leadsto \mathit{CMOBILE}2$ ♦

Proof

It holds:

$\mathit{LMOBILE} \models_{\mathrm{attr}} \mathit{weight}(\mathbf{occ}(\mathit{cube}(2))]) = 2$

and

$\mathit{CMOBILE}2 \models_{\mathrm{attr}} \mathit{weight}(\mathbf{occ}(\mathit{cube}(2))]) = 2 * 2 = 4$ ♦

But *CMOBILE*2 is an implementation of *LMOBILE* from our intuitive notion of behaviour since the weights of the cubes and submobiles are *not* observable.

6.3 Behavioural Implementations

The small running example of the mobile shows that the specification *CMOBILE*2 is not an implementation of *LMOBILE* but it is an implementation from the specifiers point of view.

Inspecting the software engineering process for a compiler being a more realistic example, e.g. two attribute grammars can be defined:

- one attribute grammar defining the mapping from the source to the target language without optimizations and
- one attribute grammar defining the mapping from the source language to an optimized program of the target language.

The first mapping cannot be characterized either using a behavioural implementation relation, because a signature change is usually performed and therefore no implementation step is obtained.

But the second mapping characterizes a behavioural implementation relation wrt. the first compilation. Since the „behavioural semantics" of the unoptimized target program has to be equal to the „behavioural semantics" of the whole optimized target program. We will see such an example in section 8.2.

To explain the decisive facts the considerations are again based on the small mobile example.

The specification *BEHLMOBILE* can be behaviourally implemented changing the weights of the cubes and splitting the undirected balance equation into directed equations being specification *CMOBILE2* of section 4.1. This proceeding results in a mobile with the same behaviour as the original one.

This behavioural notion of implementation is formalized as the behavioural model class inclusion, i.e. a behavioural specification $ASpec_1$ is behaviourally implemented by a behavioural specification $ASpec_2$ if the behavioural model class of $ASpec_2$ restricted to the signature of $ASpec_1$ is a subset of the behavioural model class of $ASpec_1$ and the usual signature and observability inclusions hold. But the definition of behavioural class is completely different to behavioural algebraic specifications.

Definition 6.3.1 (behavioural implementation)

A behavioural attributed algebraic specification $ASpec_2$ is a *behavioural implementation* of a behavioural attributed algebraic specification $ASpec_1$ (written $ASpec_1 \rightsquigarrow_{\text{beh}} ASpec_2$), iff

$sig(ASpec_1) \subseteq sig(ASpec_2)$, $obs\text{-}sorts(ASpec_1) \subseteq obs\text{-}sorts(ASpec_2)$,
$obs\text{-}attrs(ASpec_1) \subseteq obs\text{-}attrs(ASpec_2)$ and
for all behavioural models $B \in Beh(ASpec_2)$ holds $<B|_{sig(ASpec_1)}> \in Beh(ASpec_1)$. ♦

The intuitive notion of behaviour states already the proof theoretical characterization of the implementation relation: The solutions of the abstract specification has to be a subset of the solutions of the concrete one (if we do not consider the behavioural equivalence on the attribute values):

Theorem 6.3.2 (proof theoretical characterization)

Let $ASpec_1 = <\Sigma_1, F_{Attr_1}, S_{Obs_1}, F_{AttrObs_1}, Ax_1>$ and $ASpec_2 = <\Sigma_2, F_{Attr_2}, S_{Obs_2}, F_{AttrObs_2}, Ax_2>$ be beh. attributed algebraic specifications. To prove that $ASpec_1$ is implemented by

$ASpec_2$ it is sufficient to show

$Solutions(ASpec_1) \subseteq Solutions(ASpec_2)$ and $AlgAx(Ax_2) \models_{beh} AlgAx(Ax_1)$

under the assumption that $sig(ASpec_1) = sig(ASpec_2)$. The inclusion of the solution is defined as

$Solutions(ASpec_1) \subseteq Solutions(ASpec_2)$ iff
$\forall\, sol_1 \in Solutions(ASpec_1).\ \exists\, sol_2 \in Solutions(ASpec_2).\ \exists\, \sigma \in Subst.\ sol_1\sigma = sol_2.$ ♦

Proof (sketch)

$Solutions(ASpec_1) \subseteq Solutions(ASpec_2)$ and $AlgAx(Ax_2) \models_{beh} AlgAx(Ax_1)$ imply

$$\{ A \in Alg(\Sigma_{2Occ}) \mid A \models_{attr} C_2, A \models_{beh} AlgAx(Ax_2), A \models_{battr} sol_2, sol_2 \in Solutions(ASpec_2) \} \subseteq$$
$$\{ A \in Alg(\Sigma_{1Occ}) \mid A \models_{attr} C_1, A \models_{beh} AlgAx(Ax_1), A \models_{battr} sol_1, sol_1 \in Solutions(ASpec_1) \}$$

and therefore $Beh(ASpec_2) \subseteq Beh(ASpec_1)$. ♦

If we consider moreover the behavioural equivalence on the attribute values, the following characterization is obtained:

Theorem 6.3.3 (proof theoretical characterization)

Let $ASpec_1 = <\Sigma_1, F_{Attr1}, Ax_1>$ and $ASpec_2 = <\Sigma_2, F_{Attr2}, Ax_2>$ be attributed algebraic specifications. To prove that $ASpec_1$ is implemented by $ASpec_2$ it is sufficient to show

$AlgAx(Ax_2) \models_{beh} AlgAx(\sigma(Ax_1))$ and
$Solutions(ASpec_2) \cup AlgAx(Ax_2) \models_{battr} Solutions(ASpec_1)$

for all ground substitutions σ over $sig(ASpec_1)$
under the assumption that $sig(ASpec_1) \subseteq sig(ASpec_2)$. ♦

Proof

Analogous to the proof of theorem 6.2.1. The second condition is necessary to obtain the behavioural class inclusion, since in the beh. class as well the axioms of the algebraic specification as the solutions have to be valid. ♦

These theorems give characterizations for showing the correctness of implementation relations. An example for a behavioural implementation proof is discussed in the compiler case study (application of 6.3.3) and in the following section (application of 6.3.2).

6.4 Example of a Behavioural Implementation Proof

In this section we show the correctness of the behavioural implementation of *BEHLMOBILE* by *CMOBILE2*.

Theorem 6.4.1

It holds:

$BEHLMOBILE \rightsquigarrow_{beh} CMOBILE2$ ♦

Proof

According to the proof theoretical characterization we have to show:

$Solutions(ASpec_1) \subseteq Solutions(ASpec_2)$

We rewrite this condition to

for all $T \in part((T_{\Sigma}^{occ_s}(SV))_{s \in S_{Obs}})$. $Solutions^{ASpec_1}(T) \subseteq Solutions^{ASpec_2}(T)$

such that

$fact(T) = \{ T_1, T_2, \ldots, T_n \}$ iff
$T = \bigcup_{1 \le i \le n} T_i$ and $\forall\ 1 \le i \le n.\ \exists\, t \in ST_{(S,\, C,\, \varnothing)}(SV).\ \forall\, t_i \in T_i.\ Term(t_i) = t$

We show this new property with a complete set of occurrence terms. The proof is given in Appendix B.2. ♦

6.5 Properties of the Implementation Relations

In this section properties of the presented implementation relations for attributed algebraic specifications are considered, namely the transitivity of the implementation relation and the monotonicity of the implementation relations wrt. the specification building operations.

The implementation relations are transitive:

Theorem 6.5.1

The implementation relation $\rightsquigarrow$ and $\rightsquigarrow_{\text{beh}}$ are transitive. ♦

Proof

Let $ASpec_1$, $ASpec_2$ and $ASpec_3$ be attributed algebraic specifications with $ASpec_1 \rightsquigarrow ASpec_2$ and $ASpec_2 \rightsquigarrow ASpec_3$.

Obvious: $sig(ASpec_1) \subseteq sig(ASpec_3)$

Let $A \in Mod(ASpec_3)$. $<A\big|_{sig(ASpec_2)}> \in Mod(ASpec_2)$ is valid, since $ASpec_2 \rightsquigarrow ASpec_3$.
Furthermore, $<(<A\big|_{sig(ASpec_2)}>)\big|_{sig(ASpec_1)}> \in Mod(ASpec_1)$ holds, since $ASpec_1 \rightsquigarrow ASpec_2$.
Since $sig(ASpec_1) \subseteq sig(ASpec_2)$ follows: $<A\big|_{sig(ASpec_1)}> \in Mod(ASpec_1)$.
Thus it holds for all $A \in Mod(ASpec_3)$ $<A\big|_{sig(ASpec_1)}> \in Mod(ASpec_1)$ and therefore $ASpec_1 \rightsquigarrow ASpec_3$.

The transitivity of $\rightsquigarrow_{\text{beh}}$ can be shown analogously. ♦

Lemma 6.5.2

The implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{\text{beh}}$ are monotone wrt. *sum*, i.e.

$$\frac{ASpec_1 \rightsquigarrow ASpec_2 \qquad ASpec_3 \rightsquigarrow ASpec_4}{ASpec_1 + ASpec_3 \rightsquigarrow ASpec_2 + ASpec_4} \text{ and}$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2 \qquad ASpec_3 \rightsquigarrow_{beh} ASpec_4}{ASpec_1 + ASpec_3 \rightsquigarrow_{beh} ASpec_2 + ASpec_4}$$

especially holds

$$\frac{ASpec_1 \rightsquigarrow ASpec_2}{ASpec_1 + ASpec_3 \rightsquigarrow ASpec_2 + ASpec_3} \text{ and }$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2}{ASpec_1 + ASpec_3 \rightsquigarrow_{beh} ASpec_2 + ASpec_3}. \qquad \blacklozenge$$

Proof

Let $ASpec_1 \rightsquigarrow ASpec_2$, i.e. it holds $\forall\, A \in Mod(ASpec_2).\ <A|_{sig(ASpec_1)}> \in Mod(ASpec_1)$ and let $ASpec_3 \rightsquigarrow ASpec_4$, i.e. it holds $\forall A \in Mod(ASpec_4).\ <A|_{sig(ASpec_3)}> \in Mod(ASpec_3)$.

The signature inclusion is trivially valid.

Remains to show:

$\forall\, A \in Mod(ASpec_2 + ASpec_4)$ holds $<A|_{sig(ASpec_1 + ASpec_3)}> \in Mod(ASpec_1 + ASpec_3)$

Let $A \in Mod(ASpec_2 + ASpec_4)$, i.e.

$A \in Alg(sig(ASpec_2) \cup sig(ASpec_4))$ and $A|_{sig(ASpec_2)} \in Mod(ASpec_2)$
and $A|_{sig(ASpec_4)} \in Mod(ASpec_4)$

since $\forall\, B \in Mod(ASpec_2)$ holds $<B|_{sig(ASpec_1)}> \in Mod(ASpec_1)$ and
$\forall\, B \in Mod(ASpec_4)$ holds $<B|_{sig(ASpec_3)}> \in Mod(ASpec_3)$. It follows

$A \in Alg(sig(ASpec_2 + ASpec_4))$ and $<(A|_{sig(ASpec_2)})|_{sig(ASpec_1)}> \in Mod(ASpec_1)$
and $<(A|_{sig(ASpec_4)})|_{sig(ASpec_3)}> \in Mod(ASpec_3)$

using the knowledge that $sig(ASpec_1) \subseteq sig(ASpec_2)$ and $sig(ASpec_3) \subseteq sig(ASpec_4)$ results in

$A \in Alg(sig(ASpec_2 + ASpec_4))$ and $<A|_{sig(ASpec_1)}> \in Mod(ASpec_1)$
and $<A|_{sig(ASpec_3)}> \in Mod(ASpec_3)$

stating

$<A|_{sig(ASpec_1 + ASpec_3)}> \in Mod(ASpec_1 + ASpec_3)$

The proof is analogous for the behavioural implementation relation $\rightsquigarrow_{\text{beh}}$.

Since $ASpec_3 \rightsquigarrow ASpec_3$ and $ASpec_3 \rightsquigarrow_{beh} ASpec_3$ are trivially valid the conclusions

$$\frac{ASpec_1 \rightsquigarrow ASpec_2}{ASpec_1 + ASpec_3 \rightsquigarrow ASpec_2 + ASpec_3} \text{ and}$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2}{ASpec_1 + ASpec_3 \rightsquigarrow_{beh} ASpec_2 + ASpec_3} \text{ hold.}$$ ♦

Lemma 6.5.3

The implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{beh}$ are monotone wrt. *enrich*, i.e.

$$\frac{ASpec_1 \rightsquigarrow ASpec_2}{\textbf{\textit{enrich}}\ ASpec_1\ \textbf{\textit{by}}\ \Delta \rightsquigarrow \textbf{\textit{enrich}}\ ASpec_2\ \textbf{\textit{by}}\ \Delta} \text{ and}$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2}{\textbf{\textit{enrich}}\ ASpec_1\ \textbf{\textit{by}}\ \Delta \rightsquigarrow_{beh} \textbf{\textit{enrich}}\ ASpec_2\ \textbf{\textit{by}}\ \Delta}.$$ ♦

Proof

Follows from the lemma that the implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{beh}$ are monotone wrt. *sum* and that *enrich* is a special case of *sum*. ♦

Lemma 6.5.4

The implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{beh}$ are monotone wrt. *rename*, if σ is a bijective renaming morphism compatible with *Occ*, i.e.

$$\frac{ASpec_1 \rightsquigarrow ASpec_2}{\textbf{\textit{rename}}\ ASpec_1\ \textbf{\textit{by}}\ \sigma \rightsquigarrow \textbf{\textit{rename}}\ ASpec_2\ \textbf{\textit{by}}\ \sigma'} \text{ and}$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2}{\textbf{\textit{rename}}\ ASpec_1 \textbf{\textit{by}}\ \sigma \rightsquigarrow_{beh} \textbf{\textit{rename}}\ ASpec_2\ \textbf{\textit{by}}\ \sigma'}.$$

and σ is the restriction of σ' to $sig(ASpec_1)$. ♦

Proof

Let $ASpec_1 \rightsquigarrow ASpec_2$, i.e. $\forall A \in Mod(ASpec_2)$ holds $<A|_{sig(ASpec_1)}> \in Mod(ASpec_1)$.

The signature inclusion is trivially valid.

Remains to show

$\forall A \in Mod(\textbf{\textit{rename}}\ ASpec_2\ \textbf{\textit{by}}\ \sigma')$ holds $<A|_{\sigma}> \in Mod(\textbf{\textit{rename}}\ ASpec_1\ \textbf{\textit{by}}\ \sigma)$
with $\sigma: sig(ASpec_2) \rightarrow \Sigma$

Let $A \in Mod(\textbf{\textit{rename}}\ ASpec_2\ \textbf{\textit{by}}\ \sigma')$, i.e.

$A \in \{ B \in Alg(sig(\textbf{\textit{rename}}\ ASpec_2\ \textbf{\textit{by}}\ \sigma')) \mid B|_{\sigma'} \in Mod(ASpec_2) \}$

since $\forall B \in Mod(ASpec_2)$ $<B|_{sig(ASpec_1)}> \in Mod(ASpec_1)$ is valid. Thus it follows

$A \in Alg(\Sigma)$ and $<A|_{\sigma'}|_{sig(ASpec_1)}> \in Mod(ASpec_1)$.

using the definition of the model class for the renaming construct it follows

$<A|_{\sigma}> \in Mod(\textbf{rename}\ ASpec_1\ \textbf{by}\ \sigma)$.

The proof is analogous for the implementation relation $\rightsquigarrow_{beh}$. ♦

Lemma 6.5.5

The implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{beh}$ are monotone wrt. *export*, i.e.

$$\frac{ASpec_1 \rightsquigarrow ASpec_2}{\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_1 \rightsquigarrow \textbf{export}\ \Sigma\ \textbf{from}\ ASpec_2} \text{ and}$$

$$\frac{ASpec_1 \rightsquigarrow_{beh} ASpec_2}{\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_1 \rightsquigarrow_{beh} \textbf{export}\ \Sigma\ \textbf{from}\ ASpec_2}$$

with $\Sigma \subseteq sig(ASpec_1) \cap sig(ASpec_2)$ ♦

Proof

Let $ASpec_1 \rightsquigarrow ASpec_2$, i.e. $\forall\ A \in Mod(ASpec_2)$ holds $<A|_{sig(ASpec_1)}> \in Mod(ASpec_1)$.

We have to show:

$\forall\ A \in Mod(\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_2)$ holds $<A|_{\Sigma}> \in Mod(\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_1)$.

Let $A \in Mod(\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_2)$, i.e.

$\exists\ B \in Mod(ASpec_2).\ A|_{\Sigma} = B|_{\Sigma}$ implying

$\exists <B|_{sig(ASpec_1)}> \in Mod(ASpec_1).\ <A|_{\Sigma}> = B|_{\Sigma}$ implying

$<A|_{\Sigma}> \in Mod(\textbf{export}\ \Sigma\ \textbf{from}\ ASpec_1)$.

The proof is analogous for the behavioural implementation relation $\rightsquigarrow_{beh}$ ♦

Corollary 6.5.6

The implementation relations $\rightsquigarrow$ and $\rightsquigarrow_{beh}$ are monotone wrt. the specification building operations for attributed algebraic specifications. ♦

Proof

Follows immediately from the lemmas. ♦

Example 6.5.7

An example for applying the transitivity property is stated in the following fact:

Fact 6.5.8

It holds:

$LMOBILE \rightsquigarrow CMOBILE1$ ♦

Proof

Follows immediately from the facts $LMOBILE \rightsquigarrow CMOBILE$ and

CMOBILE $\rightsquigarrow$ *CMOBILE*1 and the transitivity of $\rightsquigarrow$. ♦

This implementation step was directly shown in section 6.2.

This corollary simplifies the software engineering process, since the implementation proof can be split in the following way. Let Sp_1, Sp_2,..., Sp_n be usual algebraic specifications and *Attr* an attribution based on the signature of these specifications.

In order to show

enrich Sp_1 **by** *Attr* $\rightsquigarrow$ **enrich** Sp_2 **by** *Attr* $\rightsquigarrow$ ... $\rightsquigarrow$ **enrich** Sp_n **by** *Attr*

it is sufficient to prove

$Sp_1 \rightsquigarrow Sp_2 \rightsquigarrow \ldots \rightsquigarrow Sp_n$

inducing

enrich Sp_1 **by** *Attr* $\rightsquigarrow$ **enrich** Sp_n **by** *Attr*.

Example 6.5.9

Let us consider the specification of a compiler. In this framework we have to deal with the problem of symbol tables. As a first approximation a symbol table can be viewed as a set of tuples (specification *TSET*) containing the necessary informations. As a next step these sets can be implemented by ordered lists (specification *OLIST*) which in turn are implemented by a hashtable (specification *HTABLE*). In order to show

enrich *TSET* **by** *Attr* $\rightsquigarrow$ **enrich** *OLIST* **by** *Attr* $\rightsquigarrow$ **enrich** *HTABLE* **by** *Attr* (*)

(whereby *Attr* defines the attribution of the complete compiler or the attribution to fill the symbol table) it is sufficient to prove

TSET $\rightsquigarrow$ *OLIST* $\rightsquigarrow$ *HTABLE*

and to induce (*) with the corollary.

7 Related Work

Since two specification formalisms are combined in the new approach its relation to both specification techniques is studied, namely to extensions and restrictions of algebraic specifications and extensions of attribute grammars. Some of the related work has been already discussed for motivation and introductory reasons and is neglected here.

7.1 Attribute Grammars

Standard extensions of attribute grammars are subsumed in the new approach, but „specification sugar“ introduced for writing specifications in a short and elegant way are more or less neglected in the new approach to concentrate on the main issues. However „specification sugar“ can be easily added to the new technique.

7.1.1 Higher-Order Attribute Grammars and Attribute Coupled Grammars

In [Vogt et al. 89; Swierstra, Vogt 91] *higher-order attribute grammars* and in [Ganzinger, Giegerich 84] *attribute coupled grammars* are discussed.

In higher-order attribute grammars abstract syntax trees are „first class citizens“, i.e. they can be the result of a (semantic) function, can be passed as attributes and can be grafted into the current tree, and can be attributed afterwards. Higher-order attribute grammars are an extension of pure attribute grammars such that for each production p_i: $X_0 \rightarrow X_1\, X_2 \ldots X_n$ a set of *non terminal attributes* is defined by $\{\, X_j \mid X_j = f(\ldots) \,\}$, i.e. an abstract syntax tree with root X_j is calculated using a semantic function f. Thus parts of a syntax tree can be computed using semantic functions. Beyond it parts of the abstract syntax tree can be stored as attribute values.

In attribute coupled grammars attributes are part of the abstract syntax tree. Terminal symbols and attributes are treated in the same way. I.e. attributes, which are associated with a non terminal X, are children of a node labelled with X and a terminal symbol T is an attribute of sort T, whose value is provided by the scanner. In [Ganzinger, Giegerich 84] a formal definition is given:

Let $G(\Sigma) = (N, T, P, S)$ and $G'(\Sigma) = (N', T', P', S')$ be grammars over a signature $\Sigma = (S, F)$, such that T and T' are a set of semantic sorts, $T \subseteq S$, N and N' are a set of syntactic sorts, $N \cap S = \varnothing$, $S \in N \cup T$ is the root sort and P is a set of syntactic functions with signature $N \rightarrow (N \cup T)^*$. Let $(G(\Sigma), \alpha)$ be an attribute grammar, where $G(\Sigma)$ is the underlying context free grammar and α is the usual association of attributes and attribute rules. Attributes may take their sorts either from S or N'. We call α an attribute coupling between G and G', and speak of G and G' as *attribute coupled grammars*, when the following restrictions are obeyed:

(1) In α, semantic attributes are calculated by functions from F, syntactic attributes by functions from P'.

(2) With the root sort of G, α associates exactly one attribute, whose sort is the root sort of G'. The value of this attribute is the result of the translation specified by α.

(3) Each syntactic attribute instance in a G-tree is used as an argument at most once.

(4) Each syntactic attribute instance in a G-tree other than that of the root node is used as an argument at least once.

[Swierstra, Vogt 91] stated that attribute coupled grammars can be considered as limited applications of higher-order attribute grammars. Therefore we can restrict our reflections on the comparison of higher-order attribute grammars with the new approach.

In attributed algebraic specifications the syntax of the context free grammar as well as the semantics of the semantic functions are specified by terms, thus attribute values can be abstract syntax trees, i.e. terms, and vice versa. Therefore higher-order attribute grammars are subsumed by attributed algebraic specifications.

The possibility to describe this connection can be visualized by the following examples:

(1) Attribute values are grafted into the abstract syntax tree as e.g.
$sv[x] = sv[f_{Attr}(sv[\mathbf{occ}(x)])]$

(2) Attribute values can store part of the syntax tree as e.g.
$f_{Attr}(sv[\mathbf{occ}(sv2[t])]) = t$

7.1.2 Tree Transformations

Tree transformation systems can be used, e.g. to describe parts of compiler optimizations. Pure attribute grammars are extended with attributed tree transformation rules, where predicates on the values of attribute occurrences specify whether a tree transformation rule may be applied. In [Alblas 89] a *conditional tree transformation rule* consists of an *input template*, describing the structure of the subtree to which the transformation is applicable, an *output template*, describing the structure of the transformed subtree, application *conditions* being predicates on the values of attribute occurrences of the input template, and rules defining the values of the attribute occurrences available before the evaluation process starts. A tree transformation is *applicable* to a subtree t_1 of an abstract syntax tree, if the input template *matches* the top of t_1, and the output template fits in the surrounding tree, i.e. if A and B label the roots of the input and output template, respectively, and $(X \rightarrow \alpha A \ \beta) \in P$ is the production applied immediately above t_1 then $X \rightarrow \alpha \ B \ \beta$ must be in P.

The example

```
transform <whilestat, while, <cond, boolconst>, do, stats, od>
cond boolval of boolconst = true
  into <loop-forever, forever, do, stats, od>
cond boolval of boolconst = false
  into <no-operation>
end
```

given in [Alblas 89] can be translated into our notion (allowing mixfix notation instead of the usual prefix-notation):

```
while boolexpr do stats od = while value(boolexpr) do stats od,
while true do stats od = loop-forever stats od,
while false do stats od = nop
```

The main differences between tree transformations and the specification of tree transformations using axioms are the use of *unification* instead of *matching* and the axioms define an equivalence relation on constructor terms on the semantical side.

7.1.3 Proof Principles for Attribute Grammars

[Katayama, Hoshino 81] present a verification technique for attribute grammars which can be used for the class of absolutely noncircular ones. The verification is done on the ordering of the dependency relation of the attributes. Firstly, they assign assertions to each non terminal, satisfying special conditions on used attributes. Secondly a set of verification conditions for each production is generated depending on its dependency graph. As a last step these proof obligations are verified.

[Courcelle, Deransart 88] extend the assertion method of [Katayama, Hoshino 81] to cyclic dependencies for proving the partial correctness of an attribute grammar relative to a specification, i.e. relative to the assertions. The proof is performed by fixpoint induction. They associate again a logical formula with each non terminal.

Attributed term induction - presented in this thesis - can be easily refined such that a mechanical support is possible [Duschl 94, Weiß 95], against [Courcelle, Deransart 88] state: „the practical usability of the proof method of Theorem (3.2.5) suffers from its theoretical simplicity“ whereby this proof method is refined afterwards, but it remains difficult to find strong enough assertions for each non terminal. The implemented system generates proof obligations which are verified by the TIP system [Fraus 94a, 94b]. The related work proves the correctness relative to a given specification, namely the assertions, whereby attributed term induction proves theorems over an attribute grammar. Performing proofs using attributed term induction often lemmas are necessary which are similar to the assertions of [Katayama, Hoshino 81; Courcelle, Deransart 88]. In the assertion method the proof is performed production local and therefore assertions have to describe informations of the context in which a non terminal appears. This context is concretely given in the new approach.

Drawbacks of [Katayama, Hoshino 81; Courcelle, Deransart 88] are: The proof principle is only usable for *directed* attribute equations, since the dependency graph is used as a basis for the proof principles. There are restrictions on the attributes allowed in the properties to be verified, namely only inherited/synthesized attributes and proper predecessor in the dependency graph can be used in the properties. No properties with remote access of attribute occurrences can be shown. Furthermore, it is not usable for implementation relations and stepwise refinement, since abstraction is not supported. Moreover, they suffer from efficient heuristics (for cutting the search space) and machine support.

They cannot be applied to attributed algebraic specification since the proof principles are based on the directed dependency graph and therefore are not usable for specifications with undirected attribute equations.

We have presented a proof principle usable for *undirected* attribute equations; with *no* restrictions on the properties. *No* invariants must be given, system support can be obtained and efficient heuristics using the attribute dependencies and signature flow analysis can be developed.

7.1.4 Object-Oriented Extensions

The main contributions in viewing attribute grammars in an object-oriented way are [Hedin 89, 92, 94].

Following [Hedin 89] nodes of an abstract syntax tree are regarded as instances of objects. The productions are viewed as classes and are hierarchical arranged in a class - subclass hierarchy. Attributes and attribute equations are defined in the objects and inherited (in the object-oriented sense) along the classification hierarchy, especially default behaviour, i.e. default attribute equations, can be specified and overwritten in specialized classes. A kind of order-sortedness is described, since the classification hierarchy defines a type system where subproductions are subtypes of their superproductions. Inheritance (in the object-oriented sense) can be expressed in our approach allowing order-sorted signatures instead of usual signatures, but resulting in unhandy explanations, especially in defining the calculi. Therefore the order-sorted approach was omitted. Another solution is the introduction of converting constructors, i.e. constructors performing a change of the sort of a term, and defining for the result sort of these converting constructors the attribute equations with the result, that the attribute equations are „inherited“ by the subsorts. On the other side the example given in section 2.4 of [Hedin 89] can be expressed by remote access of attribute values.

Furthermore, [Hedin 89] allows „specification sugar“ as e.g. predefined lists with the usual constructors and selectors, demand attributes and multiple equations of the form

```
for all sons(X) in P
  sons(X).a1 = f(...)
```

with its obvious semantics.

Following [Hedin 94] *door attribute grammars* being based in the approach of [Hedin 89] allow objects and reference attributes to be specified as parts of an attribution. The aim is to specify complex problems in a simple way with efficient incremental attribute evaluation. Door attribute grammars consist of three kinds of objects:

- syntax node objects representing instances of productions,
- semantic objects representing static semantics structures like e.g. symbol tables,
- door objects representing interface objects between the syntax node and the semantic objects.

I.e. an extended notion of syntax trees is obtained.

Reference objects can be expressed in our formalism by remote access. Large attribute values, i.e. complex data structures, are handled in door attribute grammars as follows:

- use of *references* in contrast to the break up of large values representing it as several small objects,
- *collection valued attributes*, allowing objects to be declared as members of a collection depending on some conditions,
- *constant objects*, being global objects.

But allowing references and objects identifiers destroy the declarative nature of attribute grammars. Furthermore, for node classes the usual attribute evaluation strategies can be used, whereas for door and semantics classes a manual implementation is necessary.

Collection valued attributes and constant objects can be seen as „specification sugar“, which can be added to the new specification technique. References can be expressed using remote access of attributes with the advantage of being declarative.

7.1.5 MAX System and its Theory

The ideas for starting research on the MAX system and the new approach are different.

The first one allows the definition of static and dynamics specifications without concerning correctness aspects and implementation relations. Whereas the aim of the second one reflects on the formal development of software in this area.

Nevertheless the *MAX system* (cf. e.g. [Poetzsch-Heffter 96]) and its formalism can be seen as a first step embedding attribute grammars in a functional and algebraic framework. This formalism defines concrete algebras for specifying the occurrences in a tree and the trees themselves, therefore new occurrence sort symbols are introduced beyond the usual sort symbols. It is referred to the occurrences of a term using selector functions on the arguments of a function. Attributes are viewed as functions and are specified in a functional way with an extended pattern-matching mechanism to define context dependent informations, in comparison to functional programming languages.

But proving the correctness of attribute grammars and their implementations or abstraction mechanisms are not supported. The output of the new approach can be a MAX specification from which an efficient program can be generated.

In the proofs the context is explicitly necessary to get induction assertions, which are strong enougth. The context is not explicitly given in the MAX system approach.

Furthermore, changing the grammar influences the attribution in a strong way, especially if is dealt with remote access of attributes, since the reference to remote attributes is via parent, sibling and child selectors depending extremly on the underlying grammar. In the new approach a more elegant view on remote access is taken. But the MAX specification formalism is more implementation oriented, especially since no unification is necessary.

7.1.6 Modularity and Reusability in Attribute Grammars

[Kastens, Waite 92] summarize extensions of attribute grammars [Dueck, Gormack 90; Farrow et al. 92] for increasing modularity in their framework and develop new specification principles for reusing attribute grammars. The main issue is the simplification of the specifications, i.e. especially „specification sugar“ is added. Firstly, they present techniques for remote access:

- a restricted remote access to attributes at the root of a subtree in a special context are possible, e.g. the reference to the enclosing block, i.e. attribute equations have the form
  $$f_{Attr_1}(sv_1[sv_2[\mathbf{occ}(x)]]) = f_{Attr_2}(sv_1[\mathbf{occ}(sv_2[x])])$$
  with some restrictions on sv_1, sv_2 and x.
- an attribute value is the union of the attribute values at occurrences which are descendants of the subtree rooted in the local context. Such a notation can be seen as „specification sugar“ and can be put in the specification mechanism of attributed algebraic specifications.
- attribute equations can be formulated for some iterative computation visiting nodes in (depth-first) left-to-right order, such dependencies can be expressed in attributed algebraic specifications using subterm identifiers and an attribute equation describing the leftmost occurrence.

Moreover, *symbol computation* and *inheritance* are introduced as a kind of subterm identifier and a kind of super non terminal for which attribute equations are defined and inherited to the sub non terminals. The sub non terminal can use the attribute equations of the super non terminal or can overwrite them. Inheritance can be expressed as discussed for [Hedin 89].

The notion of *cumulative attribution* allowing to write an arbitrary number of rules with the same production causes no problems in the new approach, since the attribute equations can be defined independent of the production (constructors in the new formalism) and the sum or enrich operator are allowed in structured attributed algebraic specifications to extend existing specifications.

I.e. in usual attribute grammar systems more or less „specification sugar“ is supported but no structuring mechanisms in the sense of e.g. algebraic specifications where really new specifications are obtained combining existing ones.

7.1.7 Cyclic Attribute Dependencies and Incremental Attribute Evaluation

In the new approach it is dealt with undirected attribute equations and it is assumed that the attribute dependencies are not cyclic. The reason for restricting to acyclic attribute dependencies is that an attribute evaluation strategy is developed for undirected attribute equations. Considering cyclic dependencies the same techniques can be used as suggested in [Farrow 86; Jones 90; Walz, Johnson 95]. Extending the formalism in such a way the proof principles have to be extended with a kind of fixpoint rule.

Incremental attribute evaluation is mainly interesting in efficient programming environments. Since the new approach can be seen on the specification side and prototypical implementation of systems, incremental attribute evaluation is not considered here. But the results of the attribute dependency discussion can be used to adapt the usual incremental attribute evaluation strategies to the new approach.

7.2 Algebraic Specifications

7.2.1 Grammars as Signatures and Attribute Grammars as Algebras

It is a well known fact that there exists a 1-1 correspondence between a context free grammar and a signature. This correspondence is achieved viewing the abstract syntax tree as a term over a corresponding signature [Chirica, Martin 76; Thatcher et al. 77].

Correspondence context free grammar - signature:

Let $G = (N, T, P, S)$ be a context free grammar. A class of many sorted algebras is associated with the context free grammar G in the following way:

Let

$$nonterminals: (N \cup T)^* \rightarrow N^*$$

be a function which yields a list of non terminals of a string w over $(N \cup T)^*$ in the same order, appearing in w. Each non terminal $X \in N$ is identified with a sort S_X and each production $p \in P$ of the form p_i: $X_0 \rightarrow w$ is identified with a function

$$f_{p_i}: sorts(nonterminals(w)) \rightarrow S_{X0}$$

whereby the input of *sorts* is a word of non terminals and the result is the corresponding list of sorts.

In [Chirica, Martin 79] an order-algebraic definition of attribute grammars is given. In their considerations they use explicit algebras defining the domains of the attributes and the semantic functions of an attribute grammar. They define the semantics of an attribute grammar as the set of solutions for a given derivation tree. The solution is the least fixpoint based on the Scott and Strachey approach [Scott, Strachey 71; Stoy 77].

We abstracted from the explicit definition of those algebras describing the semantic functions algebraically. Futhermore the semantics is defined as the set of algebras satisfying the attribute equations. The proof principle for attribute grammars presented there is the usual structural induction with the drawback that the attribute grammar has to be converted into synthesized form. A possibility is shown how to change attribute grammars into synthesized form.

7.2.2 Context Induction

[Hennicker 91] uses the same notion of a context. But he uses context induction in the framework of proving the correctness of behavioural algebraic implementations as implemented in the ISAR system [Bauer, Hennicker 93]. In his framework the Noetherian ordering is the syntactical subterm ordering on the contexts. When his context induction is not defined with the syntactical subterm ordering but with an arbitrary Noetherian relation, both cases are special cases. We start like in the context induction principle with the „trivial" context expressed by (1). The „trivial" context is a context of minimal depth of the insertion place for the context identifier. (2) expresses: If the correctness of the attribute property of a depth smaller than n is known, the property has to be proved for the depth $n + 1$.

7.2.3 Primitive Recursive Schemes and ASF+SDF

[Courcelle, Franchi-Zannettacci 82] introduced primitive recursive schemes (p.r.s.s) to express attribute grammars in the framework of algebraic specifications. p.r.s.s can be seen as restricted algebraic specifications. A p.r.s. has beyond the usual equations of an algebraic specification attribute equations of the form

<synthesized attribute>(<subterm>, <list of values for the inherited attributes>) = ...

i.e. the value of a synthesized attribute is expressed by the actual subterm and the inherited attributes for this subterm. In this notion the context is only expressed by the inherited attributes and more than one attribute equation of a usual attribute grammar is encoded in such an equation.

For this class of algebraic specifications the techniques of attribute grammars are adapted e.g. [Klint 93; Meulen 94; Deursen 94]. Being a good method applying the techniques of attribute grammars to algebraic specifications they have several drawbacks:

- Global attribute dependencies cannot be expressed;
- Attribute grammars must be „translated" into a primitive recursive scheme resulting in the loss of the intuitivity of attribute grammars, since several usual attribute equations are coded into one equation;
- The usual implementation relations for algebraic specifications are not usable for those specifications, since the definition of the attribute computation rules are viewed too local in sense of an implementation, i.e. new inherited attributes are not allowed to be introduced, e.g. to simplify the attribution, because this fact leads to a change of the functionality of the synthesized attribute functions;
- From a software engineering point of view the translation of an attribute grammar into an algebraic specification and afterwards the implementation of this specification can lead to a specification which cannot be translated back after the refinement into an attri-

bute grammar from which an efficient program can be generated. Moreover, the implementation construction is more difficult to describe since the intuitive notion of attribute grammars is lost.

One aim of the new approach was to overcome these disadvantages. Most of these problems can be solved using explicit the context. However, in the new approach the usual proof principles and implementation relations have also be extended.

7.2.4 Higher-Order Algebraic Specifications

Higher-order algebraic specifications are an extension of algebraic specifications allowing structured types and having predefined functions for the evaluation of functions. The basic notions of higher-order algebraic specifications are taken from [Kosiuczenko, Meinke 96] and are adapted to our notations without motivation and going into details. Only the definitions are presented.

Let B be any non empty set, the members which will be termed *basic types*, the set B being termed a *type basis*. The type hierarchy $H(B)$ generated by B is the set

$$H(B) = \bigcup_{n \in \mathbb{N} \cup \{\infty\}} H_n(B)$$

of formal expressions defined inductively by

$$H_0(B) = B$$

and

$$H_{n+1}(B) = H_n(B) \cup \{ (\sigma \rightarrow \tau), (\sigma \times \tau) \mid \sigma, \tau \in H_n(B) \}$$

Each element $(\sigma \rightarrow \tau) \in H(B)$ is termed a *function type*, $(\sigma \times \tau) \in H(B)$ is termed a *product type*.

A *type structure* S over a type basis B is a subset $S \subseteq H(B)$ which is closed under subtypes in the sense that for any $\sigma, \tau \in H(B)$, if $(\sigma \rightarrow \tau) \in S$ or $(\sigma \times \tau) \in S$ then both $\sigma \in S$ and $\tau \in S$. S is a *basic type structure* of $S \subseteq B$.

Given a type structure S, a *higher-order signature* Σ is an S-sorted signature with distinguished operation symbols for *projection* and *evaluation*.

Let S be a type structure over a type basis B. An S-typed signature $\Sigma = (S, F)$ is an S-sorted signature such that for each product type $(\sigma \times \tau) \in S$ two projection function symbols

$$(proj_{(\sigma \times \tau), \sigma}: (\sigma \times \tau) \rightarrow \sigma) \in F \text{ and } (proj_{(\sigma \times \tau), \tau}: (\sigma \times \tau) \rightarrow \tau) \in F \text{ (for short: } proj^1, proj^2)$$

exists. For each function type $(\sigma \rightarrow \tau) \in S$ we have an evaluation function symbol

$$(eval_{(\sigma \rightarrow \tau)}: (\sigma \rightarrow \tau), \sigma \rightarrow \tau) \in F \text{ (for short: } eval)$$

Let S be a type structure over a type basis B and let $\Sigma = (S, F)$ be an S-typed signature and let A be an S-sorted Σ-algebra.

A is an S-typed Σ-algebra iff for each production type $(\sigma \times \tau) \in S$ the function

$$proj^A_{(\sigma \times \tau), \sigma}: A_{(\sigma \times \tau)} \rightarrow A_\sigma \text{ and } proj^A_{(\sigma \times \tau), \tau}: A_{(\sigma \times \tau)} \rightarrow A_\tau$$

are the first and second projection function defined on each $a = (a_1, a_2) \in A_{(\sigma \times \tau)}$ by

$$proj^A_{(\sigma\times\tau),\sigma}(a) = a_1 \text{ and } proj^A_{(\sigma\times\tau),\tau}(a) = a_2$$

For each function type $(\sigma \rightarrow \tau) \in S$ we have an evaluation function

$$eval^A_{(\sigma\rightarrow\tau)}: A_{(\sigma\rightarrow\tau)}, A_\sigma \rightarrow A_\tau$$

on the function space $A_{(\sigma\rightarrow\tau)}$ defined by

$$eval^A_{(\sigma\rightarrow\tau)}\ (a, n) = a(n)$$

for each $a \in A_{(\sigma\rightarrow\tau)}$ and $n \in A_\sigma$.

Let S be a type structure over a type basis B and let Σ be an S-typed signature and let X be an S-indexed family of infinite sets of identifiers. The set $Ext = Ext_\Sigma$ of extensionality sentences over Σ is the set of all Σ-sentences of the form

$$\forall\, x \in X_{(\sigma\rightarrow\tau)}.\ \forall\, y \in X_{(\sigma\rightarrow\tau)}.\ (\forall\, z \in X_\sigma.\ eval_{(\sigma\rightarrow\tau)}(x, z) = eval_{(\sigma\rightarrow\tau)}(y, z) \Rightarrow x = y)$$

and

$$\forall\, x \in X_{(\sigma\times\tau)}.\ \forall\, y \in X_{(\sigma\times\tau)}.\ (proj_{(\sigma\times\tau),\sigma}(x) = proj_{(\sigma\times\tau),\sigma}(y) \wedge \\ proj_{(\sigma\times\tau),\tau}(x) = proj_{(\sigma\times\tau),\tau}(y)) \Rightarrow x = y$$

By a *higher-order equational specification* a pair (Σ, E) is meant consisting of an S-typed signature Σ and a set E of Σ-equations.

Let S be a type structure over a type basis B, Σ be an S-typed signature, X be an S-indexed family of infinite sets of identifiers and E be any set of higher-order equations over Σ and X. Define the class $Alg_{ext}(\Sigma, E)$ of all extensional models of E by

$$Alg_{ext}(\Sigma, E) = \{\, A \in Alg(\Sigma) \mid A \models E \cup Ext \,\}$$

The *infinitary higher-order equational calculus* has the following rules of inference:

(*refl*), (*sym*), (*trans*), (*subst*) as in the usual equational calculus with the extended term notion and

$$(proj)\frac{proj^1(t) = proj^1(r), proj^2(t) = proj^2(r)}{t = r}$$, for each t, r of the same product type

$$(\omega - ext)\frac{eval(t, u) = eval(r, u) \text{ for all terms } u}{t = r}$$

It strikes into the eye that the use of subterm identifiers describes a kind of restricted higher-order algebraic specification, but the use of subterm identifiers is on a very syntactical level, especially the *eval*-function state that two functions are equal if they are syntactical equal in the new approach. Therefore the extensionality equations have not to be considered in the new approach. Considering the definition of the valuation of subterm identifiers coincides with the interpretation of identifiers in the higher-order specification framework.

8 Case Studies

Three case studies are presented considered among others for this thesis. The first one was part of a larger case study considered in a project called „generating intelligent user interfaces“ which was supported by Siemens Corporate Research and Development (ZFE ST SN 51). It is taken from the area of user interface specifications and is more or less presented in [Bauer 95, 96]. The second case study shows the formal development of a small compiler. The third case study is a revised version and extension to implementation steps of [Bauer 94a, 94b] out of the framework of document architecture.

One application area of the new approach is the specification of the dynamics of user interfaces. The dynamics of the user interface for an ISDN telephone is described and some verification properties of the specification are shown. The validity of one property is shown using the proof principle of attributed term induction and other properties are shown using the analysis techniques for attributed algebraic specifications, namely using attributed signature flow analysis. Attributed algebraic specifications have been proven to be a good technique for specifying the dynamics of user interfaces and for proving their correctness. Moreover it is demonstrated how the dynamics of a user interface can be generated from the formal specification of the application.

The next case study is taken from the framework of compiler construction. The compilation of expressions to stack machine code and (un)optimized register code is considered. It is shown that the compilation preserves the semantics of the program. Afterwards the implementation step between the compilation to unoptimized register code and the compilation to optimized register code is shown. Note, that the first translations, namely from expressions to stack machine code and register code is not an implementation step because a signature change is performed. However the compilations to unoptimized and optimized register code describe a behavioural implementation relation.

Another typical application area is the use of the new technique in the area of document architecture. The problem of calculating the length of inner boxes, e.g. text boxes, boxes containing graphics, is considered such that a given length of the whole box, e.g. the length of a line, is reached. Implementations and attribute evaluation aspects for these specifications are taken into consideration.

These case studies explain by typical examples all aspects of the specification framework presented in this thesis.

The considered aspects of the thesis and their application in the case studies are summarized in table 2:

The *remote access* of attribute values is used for the specification of the compiler. It is remotely referred to an environment attribute storing the values of the identifiers. In the document architecture example a remote access to the global jolting factor of the boxes is applied. In the user interface example we make extensive use of the new occurrence technique and remote access and can shorten the specification from 19 (more or less a usual attribute grammar) to 7 axioms.

Undirected attribute equations and their implications to *attribute evaluation* (several correct attributions and attribute evaluation ordering) are discussed in the document architecture case study where an invariant for the length of the whole box is given.

Observability issues can be considered in all three case studies, but only in the compiler example it is explained in detail: In the user interface example the input/output of the user interface can be described using observable sorts, i.e. the abstract menu-items and the telephone numbers. In the compiler case study the generation of optimized and unoptimized code are behaviourally equivalent. An observability aspect for the box example can be the concrete layout.

Universally quantified formulae are used showing the correctness of the user interface specification, the correctness of the compilation process and the invariant of the document architecture example.

Existentially quantified formulae are used deriving intelligent help (user interface example) and for performing an inverse compilation process (compiler specification). Thus an inverse compilation process can be used to guarantee that a compiler performs the correct translation wrt. the compiler specification.

Standard implementation relations are considered in the document architecture example. *Behavioural implementation steps* are discussed in the compiler example. Here the implementation of a compiler generating unoptimized code by a compiler generating optimized code is proved.

The only considered *structuring mechanisms* are *enrich* and *sum.* Only these operations are used, because it is not presented a complete software project. But the use of structuring mechanisms for reuse are obvious.

Examples from different areas illustrate the applicability of the new technique to different problem domains.

Attributed signature flow analysis is applied for showing the reachability of the abstract menu-items in the user interface example and is implicitly used in the proofs.

considered aspects	**user interface specification**	**compiler specification**	**document architecture**
remote access of attribute values	shortening of the specification	environment for the used identifiers	global jolting factor of boxes
undirected equations			invariant: given length
several correct attributions			depending on the given length
observability issues	input/output shown to user	unoptimized/ optimized code	concrete layout of the boxes
universally quantified formulae	correctness: user interface	correctness: compilation	validity of invariant
existentially quantified formulae	deriving intelligent help	inverse compilation process	
standard implementation			refinement

considered aspects	user interface specification	compiler specification	document architecture
behavioural implementation		unoptimized by optimized code	
structuring mechanisms	enrich	enrich and sum	enrich
attributed signature flow analysis	user interface properties		

table 2: case studies show considered aspects

8.1 Specifying User Interfaces

This case study was part of a larger case study considered in a project called „generating intelligent user interfaces“, which has been supported by Siemens Corporate Research and Development (ZFE ST SN 51).

In the first subsection the use of attributed algebraic specifications for the specification of user interfaces is described following [Bauer 95]. Moreover we will see how the dynamics of a user interface can be generated from an algebraic specification, based on [Bauer 96].

Nowadays nearly every software project has to deal with the implementation of user interfaces, because the end-users of such systems are often computer novices using the program with little or less knowledge about the computer technology. But the development of a graphical user interface is not a trivial task. The implementation is a time-consuming, error-prone work to do and complex software engineering process. Morover it is a very critical point in the software engineering process, because the complete interaction between the user and the application is via the user interface. According to [Myers 88] 50-88% of the code of an interactive application is the code for the user interface. Furthermore, the result would be damnable having a correct proven application and an incorrect user interface. Therefore formal methods must be applied in the framework of user interface development to consider correctness aspects. Using formal methods allows the generation of user interfaces out of a declarative description (model) of the properties of an interactive application. This fact allows to enter into competition with other software developers, since the price for individual software should be low and generating software is cheaper than programming code. These generation aspects can be found in the model based user interface tools (e.g. [Bodart et al. 94; Balzert 93, 94, 95; Janssen et al. 93, 91, 93; Schreiber 94a, 94b]). We will see how the dialogue description can be generated from a formal specification of the application. The new specification formalism can be used (under some restrictions)[16] as an input for the system presented in [Schreiber 94a, 94b, 96] for the generation of a presentation and dialogue control component of an interactive system.

[16]. undirected attribute equations, algebraic specification in a functional way.

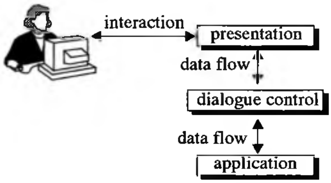


figure 21: visualization of the three layers of a program with a user interface

Considering a whole application with a user interface three layers have to be distinguished (see figure 21):
- The specification of the *presentation* (layout) the user is interacting with.
- The specification of the *dialogues* or *tasks* (dynamics) describing all possible dialogues, independent of the layout (as in [Eickel 90] for document architecture systems).
- The specification of the *application* (functional core) offering an appointed functionality which must be supported by the user interface.

Firstly this section focuses on the formal specification of dialogues and its effects on the application. Therefore it is possible to prove properties not only about the dialogues but also about its effects on the application. Formal grammars were already used in [Reisner 81] for the description of dialogues. [Hoppe 88; Payne 85; Tauber 90] developed (extended) task action grammars, which are special cases of attribute grammars, encoding more semantical informations in grammars.

The aspects presented in the first part of this section are: Using attributed algebraic specifications to link the dialogue description to application, application of the proof principle of section 5.2.2 and the analysing technique of section 3.6 to user interface verification.

Interesting correctness aspects for user interface specifications are:
- does the dialogue description offer all (exported) application functions,
- is it possible to perform an action at all,
- is an action reachable from another action,
- does the application have a given state before/after a special action is performed,
- does a special property hold before an action is performed,
- does the dialogue description ensure the applicability of an application function,
- does the validity of local context conditions result in the validity of global context conditions.

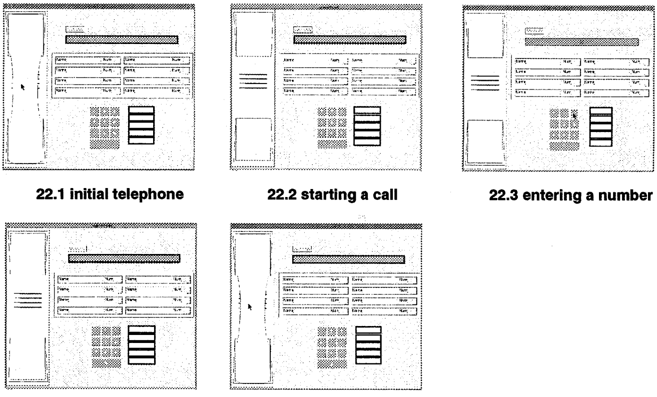
The first three items can be shown using signature flow analysis and the other items using attributed term induction.

In the last subsection we will show how the dialogue description can be generated from the algebraic specification of the application.

8.1.1 Intuitive Access to the Specification of User Interfaces

Performing a call with a user interface of an ISDN telephone is shown in figure 22. The session starts with the initial telephone (22.1). Clicking on the handset starts a telephone call. Now a telephone number, e.g. 2021, is entered (22.2, 22.3). Afterwards it can be talked with the called person (22.4). To end the call it must be clicked on the handset place

(22.5).



22.1 initial telephone

22.2 starting a call

22.3 entering a number

22.4 talking

22.5 ending a call

figure 22: making a telephone call

Viewing the telephone call in an abstract way the sentence

CALL 2021 END

was built such that *CALL* is the token yielded from the presentation clicking on the handset, *2021* entering the phone number and *END* clicking on the handset place. Therefore we have an abstract description of our telephone call independent of the actual presentation. The distinction between the abstract specification of the dialogue and the concrete presentation (layout) allows to have one dialogue specification and several concrete user interfaces, e.g.:

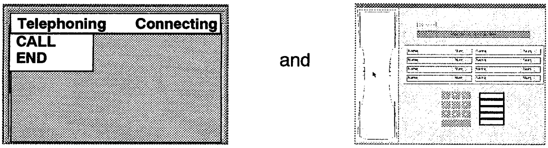


figure 23: two alternative concrete presentations

Now we can define an abstract grammar or signature for the specification of the above dialogue. A possible abstract syntax tree for the above sentence is shown in figure 24, corresponding to the term *mkCallTask*(*mkCall*(*CALL*, *mkEnterTNumber*(*2021*)), *END*):

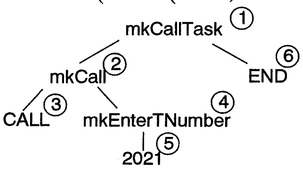


figure 24: dialogue

As a next step we associate special informations with each node of the syntax tree, i.e. ①, ②,..., ⑥, namely the attributes *statebefore* and *stateafter* describing the state of the application before and after performing a subdialogue. Attribution rules are defined to describe the effects of the dialogues on the application. Algebraic specifications are used for the specification of the functional core leading to a unifying starting point for the application and user interface development. The dialogue state of a user interface is the actual built dialogue tree and the state of the application is stored in the attribution. Proving properties of a user interface is a proof over all attributed trees. With the proof principle presented in section 5.2 it can be shown that the application has a special state at distinguished nodes. E.g. the attribute *stateafter* at node ② describes a realized telephone call.

Thus an attributed algebraic specification for describing a user interface consists of: the algebraic specification of the application (*semantics* part), the dialogue description (*syntax* part) and its effects on the application (*attribution* part). In the attribution part the calculation of the actual *layout* can be specified, too.

In the following all performable dialogues are called *dialogue state*. An element of the dialogue state is described by an attributed tree storing informations about the already performed dialogue (in the *syntax tree*) and the semantics of it (in the *attribution* of the syntax tree), i.e. the changes of the application state. The actual state is the actual attributed tree. We take the following view (for more details see the example below):

- The end-user's interactions, e.g. the selection of a menu-item, produce a stream of tokens changed into a tree by a parser.
- The actual state of the application is handled as an attribute describing how the user interactions change the application.

8.1.2 User Interface Specification of an ISDN Telephone

We start with the algebraic specification *ISDN-Application* of the application. The specification of the ISDN telephone is an enrichment of the natural numbers (*NAT*). The sorts describe the connection with a participant (*Connection*), the internal state of the telephone (*State*) and the state of a connection (*CState*). The internal state is viewed in an abstract way, i.e. two connections can be achieved with the telephone (*mkState*). *mtCon* states the empty connection. A (non-empty) connection consists of a telephone number and the status of the line (*mkCon*). A line can either be *waiting* or *telephoning*. The function *call* describes the telephone call with a single participant, *secondCall* starts a telephone call with a second participant and the *conference* function enables a conference session between the user of the telephone and the two participants on the other lines. All telephone calls are ended with *endCalls*. A telephone number is a natural number which is the observable sort.

```
spec ISDN-Application =
  enrich NAT by
    sorts Connection, CState, State
    obs-sorts Nat
    cons
      mkState: Connection, Connection → State,
      mtCon: → Connection,
      mkCon: Nat, CState → Connection,
      waiting, telephoning: → CState
    opns
```

```
      call: Nat, State → State,
      secondCall: Nat, State → State,
      conference: State → State,
      endCalls: State → State
    axioms for all nr, nr2: Nat; s: State.
      call(nr, s) = mkState(mkCon(nr, telephoning), mtCon),
      secondCall(nr, call(nr2, s)) = mkState(mkCon(nr2, waiting), mkCon(nr, telephoning)),
      conference(secondCall(nr, call(nr2, s))) =
        mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning)),
      endCalls(s) = mkState(mtCon, mtCon)
  endspec
```

The following simplifications are made:
- each telephone call is realized,
- telephone numbers are denoted by natural numbers and
- no switching between two participants is possible.

As a next step we specify the possible dialogues of the telephone.

```
  aspec ISDN-Dialogue =
   enrich NAT by
    sorts   Dialogue, Task, Call, SecondCall, Conference, EnterTNumber, CallMenu,
            SecondCallMenu, ConferenceMenu, EndMenu
    obs-sorts CallMenu, SecondCallMenu, ConferenceMenu, EndMenu
    cons
      mtDialogue: → Dialogue,
      mkDialogue: Task, Dialogue → Dialogue,
      mkCallTask: Call, EndMenu → Task,
      mkConferenceTask: Conference, EndMenu → Task,
      mkCall: CallMenu, EnterTNumber → Call,
      mkSecondCall: SecondCallMenu, EnterTNumber → SecondCall,
      mkConference: Call, SecondCall, ConferenceMenu → Conference,
      mkEnterTNumber: Nat → EnterTNumber,
      CALL: → CallMenu,
      SECONDCALL: → SecondCallMenu,
      CONFERENCE: → ConferenceMenu,
      END: → EndMenu
   endspec
```

A dialogue can be seen as a sequence of tasks. Therefore a dialogue is either an *mtDialogue* or a *Task* followed by a *Dialogue*. The task of a telephone call consists of the telephone call and the ending of the call (*mkCallTask*). The task of the conference session consists of performing the conference call and ending the calls (*mkConference*). To perform either a call or a call with a second participant an abstract menu-item[17] must be selected and afterwards the telephone number must be entered (*mkCall*, *mkSecondCall*, *mkEnterTNumber*). In the case of a conference the abstract *CONFERENCE*-menu must be selected after performing the first and second call. All calls are ended with the abstract menu-item *END*. The abstract menu-items are *CALL*, *SECONDCALL*, *CONFERENCE* and *END* and are characterized by the observable sorts.

The link between the application and the dialogue is defined using attribution:

[17] By an „abstract menu-item“ we mean a token which is yielded from the concrete representation, i.e. an abstract menu-item can be e.g. a „concrete“ menu-item, a pushbutton or a clicking on the handset in our direct manipulation user interface.

```
aspec ISDN-Attribution =
  enrich ISDN-Application + ISDN-Dialogue by
    attrs inh     statebefore: Dialogue → State
          synth   stateafter: Dialogue → State
    axioms for all d: Dialogue; t: Task; c: Call; sc: SecondCall; em: EndMenu;
          cm: ConferenceMenu; n: Nat.
    (1)   statebefore(occ(d)) = mkState(mtCon, mtCon),

    (2)   stateafter(sv[occ(mtDialogue)]) = statebefore(sv[occ(mtDialogue)]),

    (3)   stateafter(sv[occ(CONFERENCE)]) =
            conference(statebefore(sv[occ(CONFERENCE)])),
    (4)   stateafter(sv[occ(END)]) = endCalls(statebefore(sv[occ(END)])),

    (5)   statebefore(sv[mkDialogue(occ(t), d)]) = statebefore(sv[occ(mkDialogue(t, d))]),
    (6)   statebefore(sv[mkDialogue(t, occ(d))]) = stateafter(sv[mkDialogue(occ(t), d)]),
    (7)   stateafter(sv[occ(mkDialogue(t, d))]) = stateafter(sv[mkDialogue(t, occ(d))]),

    (8)   statebefore(sv[mkCallTask(occ(c), em)]) =
            statebefore(sv[occ(mkCallTask(c, em))]),
    (9)   statebefore(sv[mkCallTask(c, occ(em))]) =
            stateafter(sv[mkCallTask(occ(c), em)]),
    (10)  stateafter(sv[occ(mkCallTask(c, em))]) = stateafter(sv[mkCallTask(c, occ(em))]),

    (11)  stateafter(sv[occ(mkCall(CALL, mkEnterTNumber(n)))]) =
            call(n, statebefore(sv[occ(mkCall(CALL, mkEnterTNumber(n)))])),

    (12)  statebefore(sv[mkConferenceTask(occ(c), em)]) =
            statebefore(sv[occ(mkConferenceTask(c, em))]),

    (13)  statebefore(sv[mkConferenceTask(c, occ(em))]) =
            stateafter(sv[mkConferenceTask(occ(c), em)]),
    (14)  stateafter(sv[occ(mkConferenceTask(c, em))]) =
            stateafter(sv[mkConferenceTask(c, occ(em))]),

    (15)  stateafter(sv[occ(mkSecondCall(SECONDCALL, mkEnterTNumber(n)))]) =
            secondCall(n, statebefore(sv[occ(mkSecondCall(SECONDCALL,
                mkEnterTNumber(n)))])),

    (16)  statebefore(sv[mkConference(occ(c), sc, cm)]) =
            statebefore(sv[occ(mkConference(c, sc, cm))]),
    (17)  statebefore(sv[mkConference(c, occ(sc), cm)]) =
            stateafter(sv[mkConference(occ(c), sc, cm)]),
    (18)  statebefore(sv[mkConference(c, sc, occ(cm))]) =
            stateafter(sv[mkConference(c, occ(sc), cm)]),
    (19)  stateafter(sv[occ(mkConference(c, sc, cm))]) =
             stateafter(sv[mkConference(c, sc, occ(cm))])
endspec
```

We assume with every occurrence of sort *Dialogue*, *Task*, *Call*, *SecondCall* and *Conference* the attributes *statebefore* and *stateafter*. (1) states that the value of the attribute *statebefore* is the initial state *mkState*(*mtCon*, *mtCon*) at the root of each term of sort *Dialogue*. (2) specifies that *mtDialogue* does not change the state. Selecting the abstract menu-items *CONFERENCE* and *END* change the state of the application (axiom (3) and (4)), such that the application functions *conference* and *endCalls* are called, respectively.

(5)-(7) describe the attribute dependencies at nodes marked with *mkDialogue* and (8)-(10) for *mkCallTask*. Axiom (11) calls the application function *call* as described above. (12)-(14) defines the attribution rules for *mkConferenceTask* and (16)-(19) for *mkConference*. the attribute value of the attribute *stateafter* at nodes with an arbitrary superior tree and subordinate tree of the form *mkSecondCall(SECONDCALL, mkEnterTNumber(n))* is the result of the application function *secondCall* called with the second telephone number *n* and the value of the attribute *statebefore* at the same node as arguments (15).

The actual layout can be defined in the same way. Since we are mainly interested in coupling the dialogue specification with the application, the attribution for the layout, namely two attributes describing the layout before and after a subdialogue, is omitted. Therefore changing the attribution for the layout, changes the layout independently of the application and the dialogue specification.

The specification *ISDN-Attribution* is more or less a usual attribute grammar. We present for this example a specification, which uses the new notion of terms and remote access to get a smaller specification. The new specification describes only the necessary information, i.e. in this example the function calls of an application function to the state.

We have only to decide at which nodes in the dialogue tree the application functions are envoked.

```
aspec MinimalISDN-Attribution =
  enrich ISDN-Application + ISDN-Dialogue by
    attrs inh      statebefore: Dialogue → State
          synth    stateafter: Dialogue → State
    axioms for all d: Dialogue; t: Task; c: Call; sc: SecondCall; cm:ConferenceMenu;n,n2:Nat.
    (1)    statebefore(occ(d)) = mkState(mtCon, mtCon),
    (2)    statebefore(sv[mkDialogue(t, occ(d))]) = stateafter(sv[mkDialogue(occ(t), d)]),

    (3)    stateafter(sv[mkConferenceTask(mkConference(c, sc, CONF), occ(END)]) =
             endCalls(stateafter(sv[mkConferenceTask(mkConference(c, sc, occ(CONF)),
               END)])),

    (4)    stateafter(sv[mkCallTask(c, occ(END))]) =
             endCalls(stateafter(sv[mkCallTask(occ(c), END)]))

    (5)    stateafter(sv[mkDialogue(sv1[occ(mkCall(cm, n))]), d)]) =
             call(n, statebefore(sv[occ(mkDialogue(sv1[mkCall(cm, n)], d))])),

    (6)    stateafter(sv[mkDialogue(sv1[mkCall(cm, n)], occ(mkSecondCall(sc, n2)))]) =
             secondCall(n2, stateafter(sv[mkDialogue(sv1[occ(mkCall(cm, n))],
               mkSecondCall(sc, n2))]))

    (7)    stateafter(sv[mkConference(c, sc, occ(CONF))]) =
             conference(stateafter(sv[mkConference(s, occ(sc), CONF)]))
endspec
```

The specification shows that the use of subterm identifiers and remote access of attribute occurrences shortens the writing of specifications.

8.1.3 Proving Properties of User Interface Specifications

In this subsection we apply the analysing mechanism of section 3.6 and the proof principle to prove the correctness of the user interface specification.

Analysis of Specifications

The analysis technique *attributed signature flow analysis* can be used to show several aspects of the ISDN telephone example, e.g:

- is it possible to perform an action at all (bottom-up) and
- does the dialogue description offer all (exported) application functions (bottom-up).

Moreover, it can be shown with this technique, e.g.:

- is a dialogue step already performed at a special point of the dialogue (top-down),
- is an action performable in combination with another action (context-dependent) and
- is an action reachable from another action (context-dependent)?

Reachability Problem for Menu-Items

As stated in the introduction of this case study an important property for a user interface is: do dialogues exists such that all abstract menu-items can be selected. The bottom-up signature flow analysis of this reachability problem for abstract menu-items is defined for the ISDN telephone as follows (under the assumption that the ISDN telephone has four abstract menu-items associated with the constants *CALL*, *SECONDCALL*, *CONFERENCE* and *END*):

The signature is $\Sigma = (S, C, F)$ of the attributed algebraic specification *ISDN*.

(1) the set of domains is for all sorts $s \in S$:
$D = \{ CALL, SECONDCALL, CONFERENCE, END \}$.

(2) the set of propagation functions is
$p_f = f$ if $f \in \{ CALL, SECONDCALL, CONFERENCE, END \}$ and
$p_f(SFA[s_1], SFA[s_2], ..., SFA[s_n]) = \bigcup_{1 \leq i \leq n} SFA[s_i]$, otherwise.

(3) the set of combination functions is the usual set union for all sorts $s \in S$.

(4) a set of relations is the usual set inclusion.

Especially for the sort *Dialogue* the solution is the desired set of all abstract menu-items.

This example can be proved by the system of [Duschl 94; Weiß 95].

Application Problem for the Exported Application Functions

Another interesting property to check is whether dialogues exist such that all interface application functions, i.e. all functions which have to be supported by the user interface, can be applied. In the ISDN telephone example the interface functions are the set $\{$ *call*, *secondCall*, *conference*, *endCalls* $\}$. The problem is defined as follows:

The signature is $\Sigma = (S, C, F)$ of the attributed algebraic specification *ISDN*.

(1) the set of domains is for all sorts $s \in S$:
$D = \{ call, secondCall, conference, endCalls \}$

(2) the set of propagation functions is

$p_f(SFA[s_1], SFA[s_2],..., SFA[s_n]) = used(f) \cup \bigcup_{1 \le i \le n} SFA[s_i]$

such that *used(f)* computes the interesting application functions used in the attribution of the function *f* with $(f: s_1, s_2,\dots, s_n \rightarrow s) \in C$.

(3) for all sorts $s \in S$ the set of combination functions is the usual set union.

(4) a set of relations the usual set inclusion.

The solution can be obtained using the presented algorithm of section 3.6 resulting in the set *D* for the rootsort *Dialogue*, i.e. dialogues exist such that all exported functions can be applied.

Proving Occurence Properties

The properties which can be shown using the proof principle of attributed term induction are, e.g.:

- does the application have a given state before/after a special action is performed,
- does the dialogue description ensure the applicability of an application function,
- does the validity of local context conditions result in the validity of global context conditions,
- is a special property valid before an action is performed?

For the telephone specification the first two items are shown by an example.

An occurrence property *P* (see section 5.2.2) for occurrence terms of sort *s*, i.e. of the form $c[occ_s(t)]$ for some context $c[z_s]$ and some term *t* of sort *s*, is a formulae over the attribute occurrences of sort *s* and the semantic functions of the attributed algebraic specification, describing dependencies between attribute occurrences at these nodes. In the framework of user interface verification „dependencies between attribute occurrences at these nodes“ can be interpreted as „the application has a special state after/before a distinguished subdialogue“, or „the inputs satisfy special restrictions“.

In the telephone example we prove that after selecting the abstract conference menu-item, the state of the application is the telephoning of all three participants. Beyond it the parameter restriction for the application function *conference* is satisfied. This property can again be shown using the system of [Duschl 94; Weiß 95].

Mathematically:

Fact 8.1.3.1

For all occurrence terms *t* of sort *ConferenceMenu* holds:

$stateafter(t) = mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning))$

and

$statebefore(t) = mkState(mkCon(nr, waiting), mkCon(nr2, telephoning))$

for some telephone numbers *nr* and *nr*2. This formula is the occurrence property *P*(*t*).

Proof

We use attributed term induction for the proof of the theorem. It can be alternatively shown using a complete set of occurrence terms.

In the *base of the context induction* all minimal outer contexts with context identifier $z_{ConferenceMenu}$ and root sort *Dialogue* has to be constructed. Therefore all functions with result sort *Dialogue* have to be considered being

mkDialogue: *Task, Dialogue* → *Dialogue*

Now for all argument sorts, i.e. *Task* and *Dialogue*, a nested context induction has to be performed.

First argument of mkDialogue: Task

The actual context is *mkDialogue*(z_{Task}, $x_{Dialogue}$) with new identifier $x_{Dialogue}$. In order to finish the proof successfully, a more general context is used[18], namely $c[z_{Conference}]$ for an abstract context c (no proof is given)[19].

The sort of the property *ConferenceMenu* is not identical with *Conference*, thus a nested context induction for the construction of the minimal outer context is done. All functions with result sort *Conference* have to be considered in the actual context being

mkConference: *Call, SecondCall, ConferenceMenu* → *Conference*

For the first two argument sorts no contexts $c'[z_{ConferenceMenu}]$ of sort *Call* and *SecondCall*, respectively, exist (proof omitted)[20].

But with $c[mkConference(x_{Call}, x_{SecondCall}, z_{ConferenceMenu})]$ the minimal context is reached. Now a term induction on sort *ConferenceMenu* has to be done. The function to consider is *CONFERENCE* being a constant.

Thus we have to show

$$P(\sigma(c[mkConference(x_{Call}, x_{SecondCall}, \mathbf{occ}(CONFERENCE)]))$$

holds for all ground substitutions σ, since we are only interested in ground occurrence terms resulting in the proof obligation:

$$P(c[mkConference(mkCall(CALL, mkEnterTNumber(\alpha1_{Nat})),$$
$$mkSecondCall(SECONDCALL, mkEnterTNumber(\alpha2_{Nat})), \mathbf{occ}(CONFERENCE))])$$

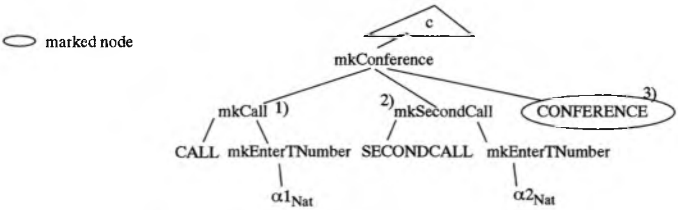
for some constants $\alpha1_{Nat}$ and $\alpha2_{Nat}$ (used as generalizations).

Visualizing the occurrence term the proof obligation *P* has to be valid at the marked node of the tree in figure 25. Looking at the attribution implies the validity of the property *P*.

[18] As usual when doing induction proofs generalizations are necessary in order to obtain finite proofs or induction assertions which are general enough to finish the proof successfully. Here the problem of finding an appropriate induction assertion appears in choosing a suitable context before doing a nested context induction.

[19] It is a kind of signature flow analysis problem presented in section 3.6. It must be shown that on every path from the root to a node of sort *ConferenceMenu* there is a node of sort *Conference*.

[20] Again a kind of signature flow analysis (cf. section 3.6).



1) stateafter = mkState(mkCon($\alpha 1_{Nat}$, telephoning), mtCon)
2) statebefore = mkState(mkCon($\alpha 1_{Nat}$, telephoning), mtCon)
stateafter = mkState(mkCon($\alpha 1_{Nat}$, waiting), mkCon($\alpha 2_{Nat}$, telephoning))
3) statebefore = mkState(mkCon($\alpha 1_{Nat}$, waiting), mkCon($\alpha 2_{Nat}$, telephoning))
stateafter = mkState(mkCon($\alpha 1_{Nat}$, telephoning), mkCon($\alpha 2_{Nat}$, telephoning))

figure 25: proof obligation

Second argument of mkDialogue: Dialogue

In this case again the context $c[z_{Conference}]$ is used for the nested context induction being a generalization of the context $mkDialogue(x_{Task}, z_{Dialogue})$. Therefore the same proof as for the first argument is obtained.

In the *context induction step* we start with an arbitrary context $c_2[z_{ConferenceMenu}]$. Since this context cannot be extended to a context $c_3[z_{ConferenceMenu}]$ which can be written as $c_2[c_4[z_{ConferenceMenu}]]$ for some non-trivial context $c_4[z_{ConferenceMenu}]$, in the context induction step nothing has to be proven. ◆

In the ISDN telephone example we have shown that after selecting the conference menu-item, the state of the application is the telephoning of all three participants and the parameter restriction of the application function *conference* is satisfied.

8.1.4 Deriving Intelligent Help

Attributed narrowing presented in subsection 5.3.2 can be used to derive intelligent help in the framework of user interface specification. Having a given state of the application and an already performed dialogue the necessary steps to reach another state of the application can be determined. Moreover, the system automatically performs the steps to reach the goal.

Let us consider as an example that a user of our ISDN telephone has started a usual telephone call with one participant and the aim is to reach a conference session.
A formalization of this fact looks like:

$$\exists\, sv_{Call,\, Task \rightarrow Task}.\ \exists\, cm_{ConferenceMenu}.$$
$$\mathit{stateafter}(\mathit{mkDialogue}(\mathit{sv}[\mathbf{occ}(\mathit{mkCall}(\mathit{CALL}, \mathit{mkEnterTNumber}(\mathit{nr}))), \mathit{cm}])) = \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mtCon})$$

$$\wedge\ \mathit{stateafter}(\mathit{mkDialogue}(\mathit{sv}[\mathit{mkCall}(\mathit{CALL}, \mathit{mkEnterTNumber}(\mathit{nr})), \mathbf{occ}(\mathit{cm})])) = \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mkCon}(\mathit{nr2}, \mathit{telephoning}))$$

A possible derivation resulting in a minimal solution, i.e. in such a solution that a minimal number of dialogue steps have to be performed, can be obtained as follows.

The start configuration is

$$\begin{array}{l}
(\{\ x1 = \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mtCon}),\\
\qquad x2 = \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mkCon}(\mathit{nr}2, \mathit{telephoning}))\ \},\\
\ [\,],\\
\ \{\ x1 = \mathit{stateafter}(\mathit{mkDialog}(\mathit{sv}[\mathbf{occ}(\mathit{mkCall}(\mathit{CALL}, \mathit{mkEnterTNumber}(\mathit{nr}))), \mathit{cm}])),\\
\qquad x2 = \mathit{stateafter}(\mathit{mkDialog}(\mathit{sv}[\mathit{mkCall}(\mathit{CALL}, \mathit{mkEnterTNumber}(\mathit{nr})), \mathbf{occ}(\mathit{cm})])\}\ )
\end{array}$$

and after several derivation steps the following configuration is obtained (see Appendix B.3):

$$\begin{array}{l}
(\{\ x2 = \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mkCon}(\mathit{nr}2, \mathit{telephoning})),\\
\qquad x2 = \mathit{conference}(x3),\\
\qquad x3 = x5,\\
\qquad x5 = \mathit{secondCall}(x6, x1)\ \},\\
\ [\ \mathit{cm}\ /\ \mathit{CONFERENCE},\\
\qquad \mathit{sv}\ /\ \mathit{sv}1[\mathit{mkConference}(z_{\mathit{Call}}, \mathit{mkSecondCall}(\mathit{SECONDCALL},\\
\qquad\qquad\qquad \mathit{mkEnterTNumber}(x8)), z_{\mathit{ConferenceMenu}})],\\
\qquad x4\ /\ \mathit{mkSecondCall}(\mathit{SECONDCALL}, \mathit{mkEnterTNumber}(x8)),\\
\qquad x7\ /\ \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mtCon}),\\
\qquad x1\ /\ \mathit{mkState}(\mathit{mkCon}(\mathit{nr}, \mathit{telephoning}), \mathit{mtCon})\ ],\\
\ \{\ \ldots\ \}\ )
\end{array}$$

As next steps the attribute equations have to be solved without complicated transformations resulting in the solution:

$$\begin{array}{l}
[\ \mathit{cm}\ /\ \mathit{CONFERENCE},\\
\qquad \mathit{sv}\ /\ \mathit{sv}1[\mathit{mkConference}(z_{\mathit{Call}}, \mathit{mkSecondCall}(\mathit{SECONDCALL},\\
\qquad\qquad\qquad \mathit{mkEnterTNumber}(\mathit{nr}2)), z_{\mathit{ConferenceMenu}})]\ ]
\end{array}$$

8.1.5 More about User Interfaces: Generating User Interfaces from Formal Specifications of the Application

This section is mainly based on [Bauer 96].

Considering once again a whole application with a user interface three layers have to be distinguished:

(5) The specification of the *presentation* (layout) the user is interacting with.

(6) The specification of the *dialogues* or *tasks* (dynamics) describing all possible dialogues, in a layout-independent way.

(7) The specification of the *application* (functional core) offering an appointed functionality which must be supported by the user interface.

Taking this scheme into consideration and looking at the user interface development process it is obvious that the user interface cannot be constructed without the knowledge of the application, since the application interface, the dynamics of the user interface and the user tasks are not independent of the application, since the state of the application controls inherent the performable dialogues. Therefore it is necessary to use the application as a starting point for the user interface development.

But which description of the application should be used? An informal specification, a formal specification or the implementation of the application? Using an informal specification does not allow the use of machine supported analysis of the specification. On the other side the implementation of the application is too low-level to be considered. Furthermore the implementation of the user interface has to be done in parallel with the implementation of the functional core to finish the implementation of both at nearly the same time.

Working with a formal specification technique allows:
- computer supported analysis of the specifications,
- elucidating the problem and
- consideration of correctness aspects of the obtained software.

Thus the starting point for the user interface and the application development is the same, namely a formal specification of the application and the software construction of both can be done hand in hand. In our framework as a starting point for the generation of user interfaces, algebraic specifications of the applications are used because this technique allows the abstract specification of the application and describes the input/output behaviour. The output of the generation process are HIT specifications [Schreiber 96] used for the generation of an executable user interface with BOSS [Schreiber94a, 94b] (“*B*edien*O*berflächen*S*pezifikations*S*ystem“ the german translation of “user interface specification system). This formalism can be translated into an attributed algebraic specification. The here presented work is part of the FUSE system (*F*ormal *U*ser Interface *S*pecification *E*nvironment). The FUSE system consists of the three components BOSS [Schreiber 96], FLUID (*F*orma*l* *U*ser *I*nterface *D*evelopment) and PLUG–IN [Lonczewski 95] (*PL*an-based *U*ser *G*uidance for *I*ntelligent *N*avigation). Within the FUSE architecture, the FLUID system plays the role of a theorem prover (cf. [Bauer 95] and in this case study) and an automatic dialogue designer. This section concentrates on the generation of the formal specification of the logical user interface - called in the following often *dynamics* of the user interface - from the formal specification of the application (i.e. problem domain model and user model).

8.1.5.1 The Problem

The specification of the application is the input of the generation process and the output is a HIT specification or an attributed algebraic specification describing the possible dialogues with the user interface on a logical view. This HIT specification in connection with a given runtime system allows the prototypical development and evaluation of a user interface with BOSS.

The Starting Point

Following [Larson 92] the user interface design decision framework consists of the following five classes:
- the *structural* and *functional* decision class determine the end users’ conceptual model,
- the *dialogue* decision class determines the dialogue style and
- the *presentation* and *pragmatic* decision class determine the refinement of the end users’ conceptual model and dialogue style.

In the structural and the functional decision class the structure of the end users’ conceptual model is specified including
- the description of conceptual objects (consumed, produced and/or accessed by the end

user),
- the application functions and
- the description of constraints and relationships that hold among conceptual objects.

I.e. more or less an abstract datatype with a special observable interface is defined in the structural and functional decision class. Such an abstract datatype can be easily specified using an algebraic specification. We assume that parameter restrictions, denoted as equations, are associated with each function symbol, i.e. we have a functionality of the form

$$fct(f) = x_{f,s_1} : s_1, x_{f,s_2} : s_2, \ldots, x_{f,s_n} : s_n \,.\, Eq_f(x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n}) \rightarrow s$$

such that f is only defined if $Eq_f(x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n})$ is valid, whereby $Eq_f(x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n})$ is an equation with the only identifiers in $\{ x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n} \}$.

The sort and constructor symbols define the conceptual objects, the function symbols define the application functions, the observable sorts characterize those objects which are observable by the end-user and the parameter restrictions with the axioms describe the constraints and relationships between the conceptual objects.

The notion of algebraic specifications has to be extended by a set of distinguished function symbols applicable to the conceptual objects (called in the following *interface functions*) which have to be supported by the user interface and the sort of the application state, i.e. the sort of the terms representing the state of the functional core. The use of interface functions cannot be neglected identifying those function symbols with observable result sort as the interface function, since it would be desirable to use application functions only changing the internal state of the application. Furthermore the initial state of an application may be defined.

Note, that the meaning of the functions (by defining the semantics of the functions by axioms and parameter restrictions) is specified, but not their format or sequencing of invocation is defined.

The three important kinds of decisions made in the dialogue decision class are
- what are the units of information exchanged between the user and the application (defined by the observable sorts and the interface functions),
- how this units of information are structured into messages between the user and the application (not considered here) and
- what the appropriate sequences of message exchange are (main issue of this contribution).

The aim of the new approach is to generate the sequence of information exchanged between the user and the application, namely to automate part of the dialogue decision class.

Specification of the Application: An Example

We start with the algebraic specification *ISDN-Application* of the application of section 8.1.2. and add the parameter restrictions and interface functions, which are necessary for the generation process. In constrast to the former specifications *call*, *secondCall* and *conference* have parameter restrictions denoted by a first order formulae after *pre. emptyConnections*, *singleConnections* and *doubleConnections* are predicates stating none, one and two connections. The interface functions, i.e. the set of functions which have to be supported by the user interface are *call*, *secondCall*, *conference* and *endCalls*.

```
  spec ISDN-Application2 =
   enrich NAT by
     sorts Connection, CState, State
     obs-sorts Nat
     cons
         mkState: Connection, Connection → State,
         mtCon: → Connection,
         mkCon: Nat, CState → Connection,
         waiting, telephoning: → CState
     opns
        call: Nat, x_{call, State} : State. pre emptyConnections(x_{call, State}) = true → State,
         secondCall: Nat, x_{secondCall, State} : State.
            pre singleConnections(x_{secondCall, State}) = true → State,
         conference: x_{conference, State} : State.
            pre doubleConnections(x_{conference, State}) = true → State,
         endCalls: State → State,
         emptyConnections: State → Bool,
         singleConnections: State → Bool,
         doubleConnections: State → Bool
    interface functions call, secondCall, conference, endCalls
    axioms forall nr, nr2: Nat, s: State.
        emptyConnections(mkState(mtCon, mtCon)) = true,
        emptyConnections(mkState(mkCon(nr, cs), c)) = false,
        singleConnections(mkState(mkCon(nr, cs), mtCon)) = true,
        singleConnections(mkState(mtCon, c)) = false,
        singleConnections(mkState(mkCon(nr, cs), mkCon(nr, cs))) = false,
        doubleConnections(mkState(mkCon(nr, cs), mkCon(nr, cs))) = true,
        doubleConnections(mkState(c, mtCon)) = false,
        call(nr, s) = mkState(mkCon(nr, telephoning), mtCon),
        secondCall(nr, call(nr2, s)) = mkState(mkCon(nr2, waiting), mkCon(nr, telephoning)),
        conference(secondCall(nr, call(nr2, s))) =
             mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning)),
        endCalls(s) = mkState(mtCon, mtCon)
  endspec
```

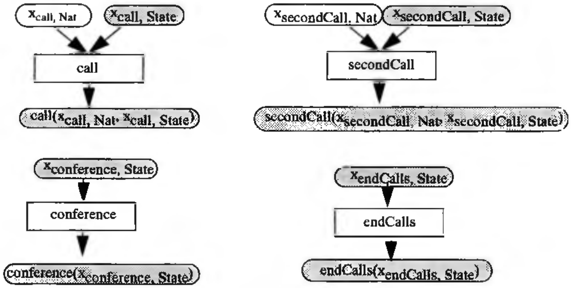
8.1.5.2 Generation Idea of the Dialogue Specification

In this section the idea for the generation of the dialogue specifications and their restrictions to different user groups are informally described.

Generation of the Dialogue Specifications

The generation process consists of several steps:

As a first step a graph is constructed with nodes marked with function symbols, identifiers for the arguments and the resulting term for each interface function. The only non-observable sort is the sort of the state of the functional core, namely *State*, marked with ▨ and observable arguments are marked with ▭ .



x_call, Nat
x_call, State
call
call(x_call, Nat, x_call, State)
x_secondCall, Nat
x_secondCall, State
secondCall
secondCall(x_secondCall, Nat, x_secondCall, State)
x_conference, State
conference
conference(x_conference, State)
x_endCalls, State
endCalls
endCalls(x_endCalls, State)

figure 26: dependency graph

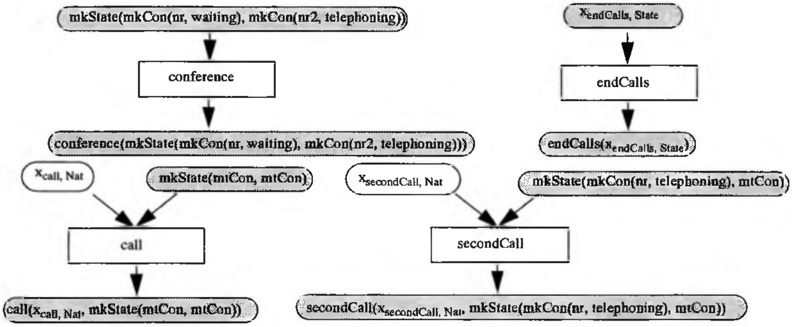
Now all the parameter restrictions of the functions can be solved using a system solving existentially quantified equations by narrowing.
Therefore the solutions of the identifiers in the parameter restrictions must be calculated, i.e. the solutions of the existentially quantified formulae:

$$\exists\, x_{call,\, State} : State.\ emptyConnections(x_{call,\, State}) = true,$$
$$\exists\, x_{secondCall,\, State} : State.\ singleConnections(x_{secondCall,\, State}) = true \text{ and}$$
$$\exists\, x_{conference,\, State} : State.\ doubleConnections(x_{conference,\, State}) = true.$$

The solutions - denoted as substitutions - can be calculated as

$$\sigma 1 = \{\ mkState(mtCon,\ mtCon)\ /\ x_{call,\, State}\ \},$$
$$\sigma 2 = \{\ mkState(mkCon(nr,\ telephoning),\ mtCon)\ /\ x_{secondCall,\, State}\ \} \text{ and}$$
$$\sigma 3 = \{\ mkState(mkCon(nr,\ waiting),\ mkCon(nr2,\ telephoning))\ /\ x_{conference,\, State}\ \}$$

These substitutions can now be applied to the graph, i.e. in the graph the identifiers $x_{call,\, State}$, $x_{secondCall,\, State}$ and $x_{conference,\, State}$ are substituted by *mkState*(*mtCon*, *mtCon*), *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*) and *mkState*(*mkCon*(*nr*, *waiting*), *mkCon*(*nr2*, *telephoning*)), respectively, resulting in:



mkState(mkCon(nr, waiting), mkCon(nr2, telephoning))
conference
conference(mkState(mkCon(nr, waiting), mkCon(nr2, telephoning)))
x_endCalls, State
endCalls
endCalls(x_endCalls, State)
x_call, Nat
mkState(mtCon, mtCon)
call
call(x_call, Nat, mkState(mtCon, mtCon))
x_secondCall, Nat
mkState(mkCon(nr, telephoning), mtCon)
secondCall
secondCall(x_secondCall, Nat, mkState(mkCon(nr, telephoning), mtCon))

figure 27: instantiated dependency graph

Since the parameter restrictions of *call* and *secondCall* influence only the second argument of sort *State* and not the first argument of sort *Nat* there is no restriction on the telephone numbers. Thus a natural number can be used as an input for the first argument of *call* and the first argument of *secondCall*. The same holds for the function *endCalls* which can be applied in every state.

The result term of the function call is *call*($x_{call,\ Nat}$, *mkState*(*mtCon*, *mtCon*)), of the function *secondCall* is *secondCall*($x_{secondCall,\ Nat}$, *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*)) and of the function conference is *conference*(*mkState*(*mkCon*(*nr*, *waiting*), *mkCon*(*nr*2, *telephoning*))). Moreover it holds

call(*nr*, *mkState*(*mtCon*, *mtCon*)) = *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*)),
secondCall(*nr*, *mkState*(*mkCon*(*nr*2, *telephoning*), *mtCon*)) =
mkState(*mkCon*(*nr*, *waiting*), *mkCon*(*nr*2, *telephoning*)),
conference(*mkState*(*mkCon*(*nr*, *waiting*), *mkCon*(*nr*2, *telephoning*)) =
mkState(*mkCon*(*nr*, *telephoning*), *mkCon*(*nr*2, *telephoning*)))
and *endCalls*(*s*) = *mkState*(*mtCon*, *mtCon*) for all States *s*.

Now the graphs can be merged together

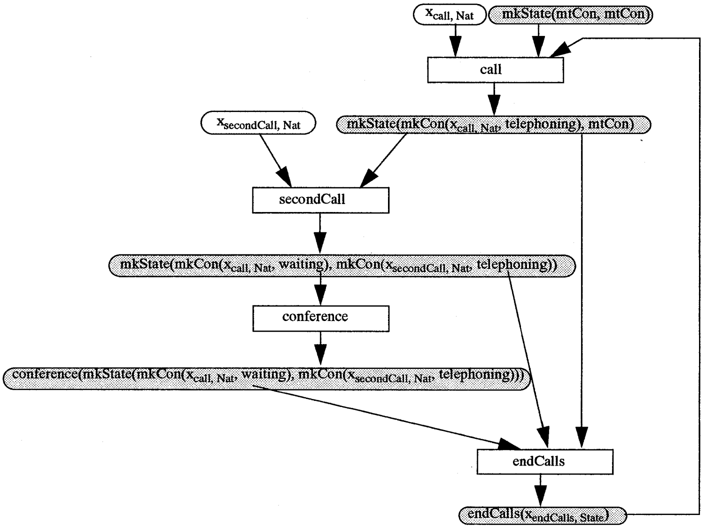

$x_{call,\ Nat}$
mkState(mtCon, mtCon)
call
$x_{secondCall,\ Nat}$
mkState(mkCon($x_{call,\ Nat}$, telephoning), mtCon)
secondCall
mkState(mkCon($x_{call,\ Nat}$, waiting), mkCon($x_{secondCall,\ Nat}$, telephoning))
conference
conference(mkState(mkCon($x_{call,\ Nat}$, waiting), mkCon($x_{secondCall,\ Nat}$, telephoning)))
endCalls
endCalls($x_{endCalls,\ State}$)

figure 28: putting the instantiated dependency graph together

and the non-observable state of the application can be omitted resulting in the graph:

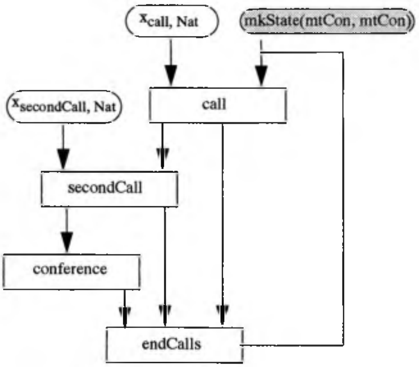


figure 29: composed instantiated dependency graph

The obtained graph can now be translated into a BOSS specification. In this generation process special dialogue style guides (specifiable in a formal way defining transformation rules for the obtained graphs) can be used, e.g. for a user or system driven dialogue style. We assume here a hard-coded transformation into the dialogue specifications.

A transaction-rule in BOSS (for more details on BOSS see [Schreiber 96]) is fired by the user, e.g. by selecting a menu-item, or by a pushbotton, i.e. each interface function is viewed as a non-repeatable transaction rule and the observable arguments as input slots, i.e. the user has to enter some information for it.

The corresponding BOSS-specification looks like

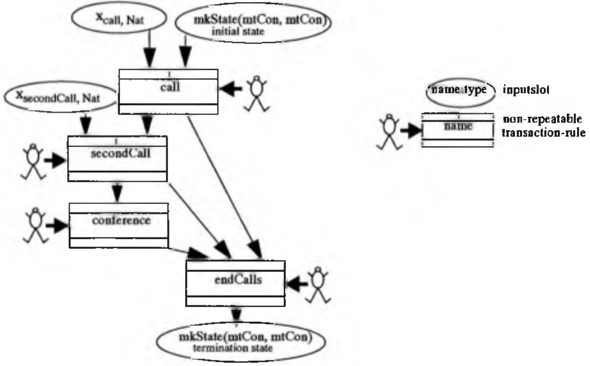


figure 30: HIT specification

Using non-repeatable transaction rules states, that the whole HIT has to be worked through starting with the initial state until the termination state is reached. Now a new instance of the HIT can be made since the termination state is equal to the initial state. this HIT specification can be translated into the presented algebraic specification *ISDN-Attribution.*

Restricting the Dialogue Specification to Different User Groups

Usual different user groups with a different functionality use a software product.

In the ISDN-example a special user group may only use the interface functions *call* and *endCalls* but not *secondCall* and *conference.*

One solution of this problem is to generate for each user group a different dialogue description, but some work has to be done twice. Therefore a more elegant way is to restrict the generated dialogue description to the interface functions of the user groups. I.e. all the nodes marked with interface functions, which are not usable by a special user group, and their argument nodes are "deleted":

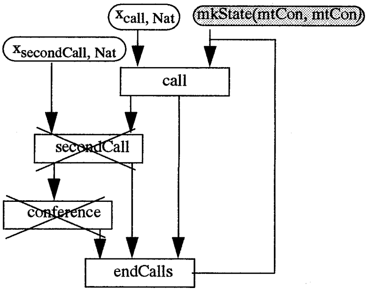


figure 31: restricting the dialogue specification to different user groups

resulting in:

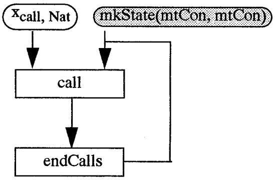


figure 32: restricted dialogue specification

with the corresponding HIT specification.

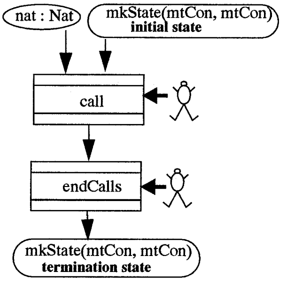


figure 33: restricted HIT specification

8.1.5.3 Generating a Specification of the Performable Dialogues

In the previous subsection we have seen by an example what the idea of generating the dialogue specification from an algebraic specification is. The starting point is a given behavioural algebraic specification $Sp = <(S, C, F), Obs, Ax>$. The sorts are split into observable and non-observable sorts and the state sort, i.e. the observable sorts describe those objects visualizable to the end-user and the non-observable objects not visible to the end-user and the objects of the state sort describe the internal state of the application also not visible to the user.

The generation process consists of 5 phases:

- Construction of the pure dependency graph,
- Solving the parameter restrictions,
- Instantiation of the pure dependency graph with the solutions of the parameter restrictions,
- Merging of the instantiated dependency graph,
- Converting the obtained graph into BOSS notation.

Construction of the pure dependency graph

The pure dependency graph $G = (N, E)$ has two kinds of nodes and edges.

For each interface function *f* with functionality

$$fct(f) = x_{f,s_1} : s_1, x_{f,s_2} : s_2, ..., x_{f,s_n} : s_n \,.\, Eq_f(x_{f,s_1}, x_{f,s_2},..., x_{f,s_n}) \rightarrow s$$

we construct the following graph $graph_f$:

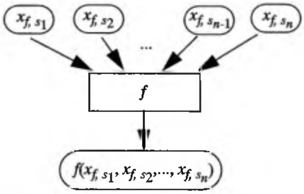


figure 34: graph of an interface function *f*

Therefore the nodes $N = N_{term} \cup N_{func}$ are split into N_{term} the set of terms and N_{func} the set of function symbols. The edges $E = E_{termtofunc} \cup E_{functoterm} \cup E_{functofunc}$ are split into edges from $n_{term} \in N_{term}$ to nodes $n_{func} \in N_{func}$ in the set $E_{termtofunc}$, edges from $n_{func} \in N_{func}$ to nodes $n_{term} \in N_{term}$ in the set $E_{functoterm}$ and edges from $n_{func} \in N_{func}$ to $n_{func} \in N_{func}$ in the set $E_{functofunc}$. $E_{functofunc}$ are used later.

The pure dependency graph is the set of graphs of each interface function *f*.

Solving the Parameter Restrictions

In this phase it is tried to solve the parameter restrictions of the interface functions, i.e. the solution of the parameter restriction for an interface function *f* with functionality

$$fct(f) = x_{f,s_1} : s_1, x_{f,s_2} : s_2, ..., x_{f,s_n} : s_\mathrm{n} \,.\, Eq_f(x_{f,s_1}, x_{f,s_2},..., x_{f,s_n}) \rightarrow s$$

are the solutions of the existential formulae:

$$\exists\, x_{f,s_1} : s_1, x_{f,s_2} : s_2, \ldots, x_{f,s_n} : s_n \,.\, Eq_f(x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n})$$

If the parameter restrictions cannot be solved at generation time (because information is missing, e.g. it is dealt with loose specifications) the run-time system of BOSS controls the parameter restrictions (therefore the parameter restrictions have to be implemented by Boolean functions).

Thus for every interface function *f* with parameter restriction the following set of solutions is obtained:

$$\sigma_f =_{\mathrm{def}} \{\ \sigma \mid Mod(Spec) \models Eq_f\,\sigma \text{ such that } \sigma \in Subst \text{ is the most general solution } \}$$

with $fct(f) = x_{f,s_1} : s_1, x_{f,s_2} : s_2, \ldots, x_{f,s_n} : s_n \,.\, Eq_f(x_{f,s_1}, x_{f,s_2}, \ldots, x_{f,s_n}) \to s$ and *Subst* is the set of all substitutions.

Instantiation of the pure dependency graph with the obtained solutions

Now for every graph $graph_f$ obtained from an interface function *f* the set of instantiated graphs $instgraph_f$ is defined by:

$$instgraph_f = \begin{cases} graph_f & \text{if } \sigma_f = \varnothing \\ \bigcup_{\sigma \in \sigma_f} \sigma(graph_f) & \text{otherwise} \end{cases}$$

such that $\sigma(graph_f)$ is defined for $graph_f$ by:

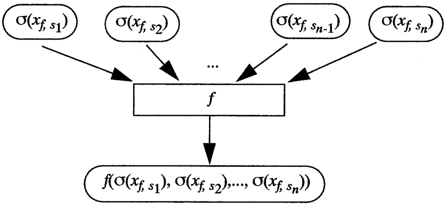


figure 35: applying a substitution to a graph

Merging of the instantiated dependency graphs

After calculating the instantiated set of graphs

$$InstGraphs = \bigcup_{f \in interface(Sp)} instgraph_f$$

whereby *interface(Sp)* calculates the interface functions of the application. The set of instantiated graphs *InstGraphs* is examined whether nodes of sort N_{term} can be connected. An edge between two nodes $t_1, t_2 \in N_{term}$ is drawn if it holds:

$Mod(Sp) \models t_1 = t_2$ can be shown and

there exists an edge $e_1 \in E_{termtofunc}$ and an edge $e_2 \in E_{functoterm}$ with

$e_1 \equiv (t_1, f_1)$ and $e_2 \equiv (f_2, t_2)$ for some function symbols f_1 and f_2.

If there is already an edge from t_2 to another term *t* of N_{term} then $instgraph_f$ is duplicated.

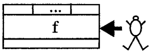
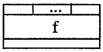
The new obtained graph is now merged together in the following way:

If there is an edge $e_1 \equiv (f_1, t_1) \in E$ and $e_2 \equiv (t_1, f_2) \in E$ then

- if there is *no* edge $e_3 \equiv (t_1, f_3) \in E$ (with $f_3 \not\equiv f_2$) then e_1 and e_2 are deleted in E and (f_1, f_2) is added to E.
- if there is an edge $e_3 \equiv (t_1, f_3) \in E$ (with $f_3 \not\equiv f_2$) then (f_1, f_2) is added to E.

Obtaining a BOSS specification

The obtained graph of the merging phase is converted into a HIT-specification as follows:

Each node $\boxed{f}$ of an interface function f is converted into a transaction rule  if f is interface functions and an equational rule  otherwise.

The restriction of the dialogue description for special user groups is done deleting the non-usable interface functions from the obtained HIT specification.

8.1.6 Remarks

We have pointed out how to specify interactive systems using attributed algebraic specifications and their verification using attributed term induction and attributed signature flow analysis. The specification formalism allows the distinction between the application and the dialogue description. The whole specification of an interactive application with a user interface can be described using the new approach, but the clear distinction between the three layers (application, user interface dynamics and concrete layout) is preserved.

Correctness aspects can only be considered in the framework of model based user interface tools [Balzert 93, 94, 95; Bodart et al. 94; Foley et al. 91, 93; Janssen et al. 93], since the layout oriented tools are too low-level. But there are model based tools which employ a specification technique with a missing logical framework, e.g. [Balzert 93, 94, 95; Bodart et al. 94; Janssen et al. 93]. Furthermore, the dialogues are sometimes specified independent of the effects on the application. Working with pre- and postconditions as in [Foley et al. 91, 93] makes the verification more difficult, since the property of the reachability test presented in this chapter is a semantical problem and not a syntactical one. Specifying dialogues in the temporal logical framework makes proving properties more complicated than reasoning in the classical logic.

Using the structuring mechanisms well-known from algebraic specifications allows to use these technique also for larger projects. The experience shows that the generation of the dialogue description for subspecifications can often be put together without considering the context in which the subspecifications are used. Otherwise normalization techniques exists for the structured algebraic specifications and the normalized specification can be used as the starting point for the generation process.

Related work on the topic of user interface generation is discussed in [Bauer 96].

8.2 Specifying Compilers

In this case study we present compiler specifications describing the translation of expressions into usual stack machine code and for the translation of the same expression language into register code. For both compilers we show the correctness of the translation. Moreover, the correctness of the behavioural implementation of a compiler for the translation of expressions into register machine code by a compiler for the translation of expressions into optimized register code is shown.

Attribute grammars are a well accepted tool for the specification of compilers. But pure attribute grammar systems suffer from the possibility to verify the compilation, i.e. to prove that the attribution yields a correct target program. In the framework of attributed algebraic specifications we can use the outlined theorem proving techniques to verify a compiler specification.

A specification of a compiler (by an attribute grammar) can be divided into three components:

- a context free grammar describing the source language,
- a context free grammar describing the target language,
- an attribute grammar with distinguished code attribute containing the compiled program.

In order to prove the correctness of the compiler, the semantics of the source and the target language have to be specified. We define the dynamic semantics of the languages in an algebraic way like shown in [Berghammer et al. 87] using already the advantages of the new technique.

8.2.1 Syntax and Semantics of the Source Language

Expressions are translated into stack machine code as described in [McCarthy, Painter 67; Berghammer et al. 87]. The syntax of the source language is the description of expressions consisting of natural numbers, identifiers having a value relative to a given environment and addition, subtraction and multiplication of expressions (*SOURCE*).

```
aspec SOURCE =
  enrich NAT + ID by
    sorts Expr, Op
    cons
      natexpr: Nat → Expr,
      idexpr: Id → Expr,
      add, sub, mult: → Op,
      comp: Expr, Op, Expr → Expr
endspec
```

Its semantics is defined in *SOURCE_SEM*.

```
aspec SOURCE_SEM =
  enrich NAT + ID + SOURCE + ENVIRONMENT by
    attrs inh     env: Expr → Env,
          synth   value: Expr → Nat
    axioms for all n: Nat; id: Id; e, e1, e2: Expr.
    (1)   value(sv[occ(natexpr(n))]) = n,
    (2)   value(sv[occ(idexpr(id))]) = lookup(id, env(occ(sv[idexpr(id)]))),
    (3)   value(sv[occ(comp(e1, add, e2))]) =
            value(sv[comp(occ(e1), add, e2)]) + value(sv[comp(e1, add, occ(e2))]),
    (4)   value(sv[occ(comp(e1, sub, e2))]) =
            value(sv[comp(occ(e1), sub, e2)]) - value(sv[comp(e1, sub, occ(e2))]),
    (5)   value(sv[occ(comp(e1, mult, e2))]) =
          value(sv[comp(occ(e1), mult, e2)]) * value(sv[comp(e1, mult, occ(e2))]),
    (6)   env(occ(e)) = givenEnv
 endspec
```

The value of a natural number is the natural number (1). The value of an identifier is its value in the given environment (2). The value of a composed binary operation is the operation applied to the value of its arguments (3)-(5). The environment at the root of an expression is a given environment (6).

8.2.2 Compiling Expressions into Stack Machine Code

Expressions are compiled such that the value of the expressions are calculated on a stack. Stacks are a syntactical enrichment of the natural numbers by the sort *Stack* and the operation *mtStack* representing an empty stack, *push* adding an element of sort *Nat* to a given stack yielding a new stack, *pop* popping the stack and *top* yielding the top element of a given stack.

```
 aspec STACK =
   enrich NAT by
     sorts Stack
     cons
       mtStack: → Stack,
       push: Nat, Stack → Stack
     opns
       pop: Stack → Stack,
       top: Stack → Nat
     axioms for all n: Nat; s: Stack.
     (1)   pop(push(n, s)) = s,
     (2)   top(push(n, s)) = n
 endspec
```

The target language of the stack machine consists of the following instructions: The empty instruction *mtInstr* does not influence the stack. *NST(n)* pushes the natural number *n* on top of the stack and *IST(id)* pushes the value of the identifier *id* on top of the stack. *ADD*, *SUB*, *MULT* adds, substracts and multiplies the two top elements of the stack, pops them and pushes the result on the stack. Instructions are concatenated with ;.

```
 aspec TARGET =
   enrich STACK + NAT + ID by
     sorts Instr
     cons mtInstr: → Instr,
           NST: Nat → Instr,
           IST: Id → Instr,
           ADD, SUB, MULT: → Instr,
           .;. : Instr, Instr → Instr
 endspec
```

The attributes are

- *value*, which is the computed value on the stack and defines the semantics of the target language,
- *stacka* describes the stack after the instructions,
- *stackb* describes the stack before the instructions and
- the attribute *env* is used as above.

The attribute equations for the root of terms of sort *Instr* denote that at the beginning the stack is empty and it is started with a given environment *givenEnv*.

```
aspec TARGET_SEM =
  enrich NAT + ID + STACK + TARGET by
    attrs synth  value: Instr → Nat,
                 stacka: Instr → Stack,
          inh    stackb: Instr → Stack,
                 env: Instr → Env
    axioms for all i, i1, i2: Instr; n: Nat; id: Id.
    (1)    stackb(occ(i)) = mtStack,
    (2)    stackb(sv[occ(i1);i2]) = stackb(sv[occ(i1; i2)]),
    (3)    stackb(sv[i1; occ(i2)]) = stacka(sv[occ(i1); i2]),

    (4)    env(occ(i)) = givenEnv,

    (5)    stacka(sv[occ(ADD)]) = push(top(pop(stackb(sv[occ(ADD)]))) +
              top(stackb(sv[occ(ADD)])), pop(pop(stackb(sv[occ(ADD)])))),
    (6)    stacka(sv[occ(SUB)]) = push(top(pop(stackb(sv[occ(SUB)]))) -
              top(stackb(sv[occ(SUB)])), pop(pop(stackb(sv[occ(SUB)])))),
    (7)    stacka(sv[occ(MULT)]) = push(top(pop(stackb(sv[occ(MULT)]))) *
              top(stackb(sv[occ(MULT)])), pop(pop(stackb(sv[occ(MULT)])))),
    (8)    stacka(sv[occ(IST(id))]) =
              push(lookup(id, env(occ(sv[IST(id)]))), stackb(sv[occ(IST(id))])),
    (9)    stacka(sv[occ(NST(n))]) = push(n, stackb(sv[occ(NST(n))])),
    (10)   stacka(sv[occ(mtInstr)]) = stackb(sv[occ(mtInstr)]),
    (11)   stacka(sv[occ(i1;i2)]) = stacka(sv[i1; occ(i2)]),

    (12)   value(sv[occ(i)]) = top(stacka(sv[occ(i)]))
endspec
```

A compiler can be formally described by a function

$$compile\colon T_{\Sigma_{source}} \rightarrow T_{\Sigma_{target}}$$

which translates programs denoted by terms over Σ_{source} into terms over Σ_{target}. The condition which has to be satisfied is

$$\forall\, t \in T_{\Sigma_{source}} .\, I_{source}(t) = I_{target}(compile(t))$$

with the semantic functions

$$I_{source}\colon T_{\Sigma_{source}} \rightarrow M \text{ and } I_{target}\colon T_{\Sigma_{target}} \rightarrow M$$

whereby M is the semantic domain of the programming languages.

The interpretation mappings I_{source} and I_{target} are specified by two attributed algebraic specifications, namely *SOURCE_SEM* and *TARGET_SEM*, with distinguished attribute *value* of sort *Nat*, which is the semantic domain of the small languages.

The function *compile* is defined by the attribute *code* of the following attributed algebraic specification *COMPILER* which is a syntactical enrichment of the attributed algebraic specification *SOURCE* and *TARGET*, i.e. the new attribute *code* and the new attribution is added to *SOURCE* and *TARGET*.

```
aspec COMPILER =
  enrich SOURCE + TARGET by
    attrs synth code: Expr → Instr
    axioms for all n: Nat; id: Id; e, e1, e2: Expr.
    (1)    code(sv[occ(natexpr(n))]) = NST(n),
    (2)    code(sv[occ(idexpr(id))]) = IST(id),
    (3)    code(sv[occ(comp(e1, add, e2))]) =
              code(sv[comp(occ(e1), add, e2)]) ; code(sv[comp(e1, add, occ(e2))]) ; ADD,
    (4)    code(sv[occ(comp(e1, SUB, e2))] =
              code(sv[comp(occ(e1), sub, e2)]) ; code(sv[comp(e1, sub, occ(e2))]) ; SUB,
    (5)    code(sv[occ(comp(e1, mult, e2))] =
              code(sv[comp(occ(e1), mult, e2)]) ; code(sv[comp(e1, mult, occ(e2))]) ; MULT
endspec
```

Example 8.2.2.1

In order to show the correctness of the compiler we have to show that

$$\forall\, e_{Expr}.\ value(\mathbf{occ}(e)) = value(\mathbf{occ}(code(\mathbf{occ}(e))))$$

Therefore the following facts are necessary:

Fact 8.2.2.2

It holds:

$$\forall\, sv_{Instr \to Instr}.\ \forall\, sv'_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$stacka(sv[\mathbf{occ}(code(sv'[\mathbf{occ}(e)]))]) =$$
$$push(value(sv[\mathbf{occ}(code(sv'[\mathbf{occ}(e)]))]),\ stackb(sv[\mathbf{occ}(code(sv'[\mathbf{occ}(e)]))]))$$

Proof

See Appendix B.4. ◆

Fact 8.2.2.3

It holds:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.\ value(sv[\mathbf{occ}(e)]) = value(\mathbf{occ}(code(sv[\mathbf{occ}(e)])))$$

Proof

This property is shown using the following complete set of occurrence terms:

$$\{\ sv[\mathbf{occ}(natexpr(n))],\ sv[\mathbf{occ}(idexpr(id))],\ sv[\mathbf{occ}(comp(e1,\ add,\ e2))],$$
$$sv[\mathbf{occ}(comp(e1,\ sub,\ e2))],\ sv[\mathbf{occ}(comp(e1,\ mult,\ e2))]\ \}$$

Proof can be found in Appendix B.4. ◆

Fact 8.2.2.4

The compiler works correct. ◆

Proof

Because of the fact. ◆

8.2.3 Compiling Expressions into Registercode

The compilation of expressions into stack machine code and its verification was already described in the mentioned literature. In this subsection we present an alternative compilation of expressions into unoptimized register code and afterwards into optimized register code, which is difficult to be formulated in algebraic specifications, since a several pass attribution is necessary in the optimized case. The idea - well known from lectures in compiler construction, see e.g. [Aho et al. 86] - is explained by the following example.

Let us consider the expression term

comp(*natexpr*(1), *add*, *comp*(*natexpr*(2), *add*, *comp*(*natexpr*(3), *sub*, *natexpr*(4))))

over the specification *SOURCE* from above which is visualized in figure 36.

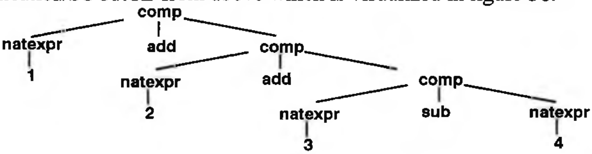


figure 36: unoptimized register code

An unoptimized code is generated if the tree is traversed in a depth first way, resulting e.g. in the following code (*R*[*i*] denotes register *i*):

```
R[1] = 1;
R[2] = 2;
R[3] = 3;
R[4] = 4;
R[3] = R[3] - R[4];
R[2] = R[2] + R[3];
R[1] = R[1] + R[2];
```

If an expression, which needs more registers for its evaluation than another expression, is calculated before the other expression, the following computation is obtained:

```
R[1] = 3;
R[2] = 4;
R[1] = R[1] - R[2];
R[2] = 2;
R[1] = R[2] + R[1];
R[2] = 1;
R[1] = R[2] + R[1];
```

In the optimized case two registers instead of four registers in the unoptimized case are needed.

Register code is defined as follows: *mtCode* is the empty register code, *regIs*(*r*, *v*) assigns to the register *r* the value *v*, *seq* concatenates two register code expressions, *ADD r r*1 *r*2 adds the values of register *r*1 and *r*2 and stores the result in register *r*. *SUB* and *MULT* are defined analogously.

```
aspec REG =
  enrich NAT by
    sorts Code
    cons  mtCode: → Code,
          regIs: Nat, Nat → Code,
          seq: Code, Code → Code,
          mkRes: Nat → Result
endspec
```

The semantics of the register code machine is specified as follows: *regval* calculates the value of a register after performing a sequence of instructions, *val* yields the value of a register depending on the used register before and *value* the value stored in register 1. .&. is the overwriting operation. *isdefined* tests whether a register is defined after a sequence of instructions is executed.

```
aspec REG_SEM =
 enrich NAT + ID + OPT + ENV + REG by
   obs-sorts Nat
   opns .&.: Nat, Nat → Nat,
          isdefined: Nat, Code → Bool.
          error: → Nat
   obs-attrs value
   attrs synth regval: Nat, Instr → Nat,
                 val: Nat → Nat,
                 value: Instr → Nat
   axioms for all n, m, r, r1, r2: Nat; i1, i2: Code; e: Expr.
   (1)    regval(r, mtInstr) = error,
   (2)    regval(r, regls(r, n)) = n,
   (3)    regval(r, ADD(r, r1, r2)) = val(r1) + val(r2),
   (4)    regval(r, SUB(r, r1, r2)) = val(r1) + val(r2),
   (5)    regval(r, MULT(r, r1, r2)) = val(r1) + val(r2),
   (6)    regval(r, seq(i1, i2)) = regval(r, i1) & regval(r, i2)

   (7)    n & m = m,
   (8)    n & error = n,
   (9)    error & error = error,

   (10)   val(usedregb(sv[occ(e)])) = regval(usedregb(sv[occ(e)]), code(sv[occ(e)])),

   (11)   value(occ(code(occ(e)))) = regval(1, occ(code(occ(e))),

   (12)   isdef(usedregb(sv[occ(e)])) = isdefined(usedregb(sv[occ(e)]), code(sv[occ(e)])),

   (13)   isdefined(r, mtInstr) = false,
   (14)   isdefined(r, regls(r, n)) = true,
   (15)   isdefined(r, ADD(r1, r2, r3)) = if eq(r, r1) then isdef(r2) and isdef(r3) else false fi,
   (16)   isdefined(r, SUB(r1, r2, r3)) = if eq(r, r1) then isdef(r2) and isdef(r3) else false fi,
   (17)   isdefined(r, MULT(r1, r2, r3)) = if eq(r, r1) then isdef(r2) and isdef(r3) else false fi,
   (18)   isdefined(r, seq(i1, i2)) = isdefined(r, i1) or isdefined(r, i2)
endspec
```

The compilation for the unoptimized case can be specified in the following way:

```
aspec NOPT =
 enrich SOURCE + REG by
   attrs synth  code: Expr → Code,
                usedrega: Expr → Nat
         inh    usedregb: Expr → Nat
   axioms for all n: Nat; id: Id; e, e1, e2: Expr.
   (1)    code(sv[occ(natexpr(n))]) = regls(usedregb(sv[occ(natexpr(n))]), n),
   (2)    code(sv[occ(idexpr(id))]) = regls(usedregb(sv[occ(idexpr(id))]), lookup(id, givenEnv)),
   (3)    code(sv[occ(comp(e1, add, e2))]) = seq(code(sv[comp(occ(e1), add, e2)]),
            seq(code(sv[comp(e1, add, occ(e2))]),
            ADD(usedregb(sv[occ(comp(e1, add, e2))]),
              usedregb(sv[occ(comp(e1, add, e2))]),
              usedregb(sv[comp(e1, add, occ(e2))])))),
    (4)     code(sv[occ(comp(e1, sub, e2))]) = seq(code(sv[comp(occ(e1), sub, e2)]),
              seq(code(sv[comp(e1, sub, occ(e2))]),
              SUB(usedregb(sv[occ(comp(e1, sub, e2))]),
                usedregb(sv[occ(comp(e1, sub, e2))]),
                usedregb(sv[comp(e1, sub, occ(e2))])))),
    (5)     code(sv[occ(comp(e1, mult, e2))]) = seq(code(sv[comp(occ(e1), mult, e2)]),
              seq(code(sv[comp(e1, mult, occ(e2))]),
              MULT(usedregb(sv[occ(comp(e1, mult, e2))]),
                usedregb(sv[occ(comp(e1, mult, e2))]),
                usedregb(sv[comp(e1, mult, occ(e2))])))),

    (6)     usedregb(occ(e)) = 1,
    (7)     usedregb(sv[comp(occ(e1), op, e2)]) = usedregb(sv[occ(comp(e1, op, e2))]),
    (8)     usedregb(sv[comp(e1, op, occ(e2))]) = usedrega(sv[comp(occ(e1), op, e2)]),

    (9)     usedrega(sv[occ(natexpr(n))]) = usedregb(sv[occ(natexpr(n))]) + 1,
    (10)    usedrega(sv[occ(idexpr(id))]) = usedregb(sv[occ(idexpr(id))]) + 1,
    (11)    usedrega(sv[occ(comp(e1, op, e2))]) = usedregb(comp(occ(e1), op, e2)
 endspec
```

For the optimized case the needed registers have to be calculated and depending on the needed registers the code is generated:

```
 aspec OPT =
  enrich SOURCE + REG by
    attrs synth   code: Expr → Code,
                  usedrega: Expr → Nat,
                  needed: Expr → Nat,
          inh     usedregb: Expr → Nat
    axioms for all n: Nat; id: Id; e, e1, e2: Expr.
    (1)     code(sv[occ(natexpr(n))]) = regls(usedregb(sv[occ(natexpr(n))]), n),
    (2)     code(sv[occ(idexpr(id))]) = regls(usedregb(sv[occ(idexpr(id))]), lookup(id, givenEnv)),
    (3)     code(sv[occ(comp(e1, add, e2))]) =
            if less(needed(sv[comp(e1, add, occ(e2))]), needed(sv[comp(occ(e1), add, e2)]))
              then  seq(code(sv[comp(occ(e1), add, e2)]),
                      seq(code(sv[comp(e1, add, occ(e2))]),
                      ADD(usedregb(sv[occ(comp(e1, add, e2))]),
                          usedregb(sv[comp(occ(e1), sub, e2)]),
                          usedregb(sv[comp(e1, add, occ(e2))]))))
              else  seq(code(sv[comp(e1, add, occ(e2))]),
                      seq(code(sv[comp(occ(e1), add, e2)]),
                      ADD(usedregb(sv[occ(comp(e1, add, e2))]),
                          usedregb(sv[comp(occ(e1), sub, e2)]),
                          usedregb(sv[comp(e1, add, occ(e2))])))) fi,
    (4)     code(sv[occ(comp(e1, sub, e2))]) =
            if less(needed(sv[comp(e1, sub, occ(e2))]), needed(sv[comp(occ(e1), sub, e2)]))
              then  seq(code(sv[comp(occ(e1), sub, e2)]),
                      seq(code(sv[comp(e1, sub, occ(e2))]),
                      SUB(usedregb(sv[occ(comp(e1, sub, e2))]),
                          usedregb(sv[comp(occ(e1), sub, e2)]),
                          usedregb(sv[comp(e1, sub, occ(e2))]))))
              else  seq(code(sv[comp(e1, sub, occ(e2))]),
                      seq(code(sv[comp(occ(e1), sub, e2)]),
                      SUB(usedregb(sv[occ(comp(e1, sub, e2))]),
                          usedregb(sv[comp(occ(e1), sub, e2)]),
                          usedregb(sv[comp(e1, sub, occ(e2))])))) fi,
(5)   code(sv[occ(comp(e1, mult, e2))]) =
      if less(needed(sv[comp(e1, mult, occ(e2))]), needed(sv[comp(occ(e1), mult, e2)]))
        then  seq(code(sv[comp(occ(e1), mult, e2)]),
                seq(code(sv[comp(e1, mult, occ(e2))]),
                MULT(usedregb(sv[occ(comp(e1, mult, e2))]),
                    usedregb(sv[comp(occ(e1), sub, e2)]),
                    usedregb(sv[comp(e1, mult, occ(e2))]))))
        else  seq(code(sv[comp(e1, mult, occ(e2))]),
                seq(code(sv[comp(occ(e1), mult, e2)]),
                MULT(usedregb(sv[occ(comp(e1, mult, e2))]),
                    usedregb(sv[comp(occ(e1), sub, e2)]),
                    usedregb(sv[comp(e1, mult, occ(e2))])))) fi,

(6)   needed(sv[occ(natexpr(n))]) = 1,
(7)   needed(sv[occ(idexpr(id))]) = 1,
(8)   needed(sv[occ(comp(e1, op, e2))]) =
      if less(needed(sv[comp(e1, op, occ(e2))]), needed(sv[comp(occ(e1), op, e2)]))
        then needed(sv[comp(occ(e1), op, e2)])
        else if less(needed(sv[comp(occ(e1), op, e2)]), needed(sv[comp(e1, op, occ(e2))]))
                then needed(sv[comp(e1, op, occ(e2))])
                else needed(sv[comp(e1, op, occ(e2))]) + 1 fi fi

(9)   usedregb(occ(e)) = 1,
(10)  usedregb(sv[comp(occ(e1), op, e2)]) =
      if less(needed(sv[comp(occ(e1), op, e2)]), needed(sv[comp(e1, op, occ(e2))]))
        then usedrega(sv[comp(e1, op, occ(e2))])
        else usedregb(sv[occ(comp(e1, op, e2))]) fi,
(11)  usedregb(sv[comp(e1, op, occ(e2))]) =
      if less(needed(sv[comp(occ(e1), op, e2)]), needed(sv[comp(e1, op, occ(e2))]))
        then usedregb(sv[occ(comp(e1, op, e2))])
        else usedrega(sv[comp(occ(e1), op, e2)]) fi,

(12)  usedrega(sv[occ(natexpr(n))]) = usedregb(sv[occ(natexpr(n))]) + 1,
(13)  usedrega(sv[occ(idexpr(id))]) = usedregb(sv[occ(idexpr(id))]) + 1,
(14)  usedrega(sv[occ(comp(e1, op, e2))]) = usedregb(sv[occ(comp(e1, op, e2))])
endspec
```

Fact 8.2.3.1

The following equation is valid in both specifications *NOPT* and *OPT*

$$\forall\ sv_{Expr \to Expr}.\ \forall\ e_{Expr}.$$
$$isdef(usedregb(sv[\mathbf{occ}(e)])) = true \qquad \blacklozenge$$

Proof

Appendix B.4 ◆

Fact 8.2.3.2

The following equation stating a connection between the specifications *SOURCE* and *NOPT* is valid:

$$\forall\ sv_{Expr \to Expr}.\ \forall\ e_{Expr}.$$
$$value(sv[\mathbf{occ}(e)]) = val(usedregb^{NOPT}(sv[\mathbf{occ}(e)])) \qquad \blacklozenge$$

Proof

Appendix B.4 ◆

Fact 8.2.3.3

The following equation stating a connection between the specifications *OPT* and *NOPT* is valid:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$val(usedregb^{OPT}(sv[\mathbf{occ}(e)])) = val(usedregb^{NOPT}(sv[\mathbf{occ}(e)])) \quad ◆$$

Proof

Appendix B.4 ◆

Fact 8.2.3.4

After performing the computation in register 1 the result of the expression is stored as well in *OPT* as in *NOPT.* ◆

Proof

Follows immediately from the fact:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$val(usedregb^{OPT}(sv[\mathbf{occ}(e)])) = val(usedregb^{NOPT}(sv[\mathbf{occ}(e)]))$$
$$\wedge\ usedregb^{OPT}(\mathbf{occ}(sv[e]) = 1 = usedregb^{NOPT}(\mathbf{occ}(sv[e]) \quad ◆$$

Fact 8.2.3.5

The following equation stating a connection between the specifications *NOPT* and *SOURCE* is valid:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$val(usedregb^{NOPT}(sv[\mathbf{occ}(e)])) = value(sv[\mathbf{occ}(e)]) \quad ◆$$

Proof

analogous to the fact

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$regval(usedregb^{OPT}(sv[\mathbf{occ}(e)])) =$$
$$regval(usedregb^{NOPT}(sv[\mathbf{occ}(e)]), code^{NOPT}(sv[\mathbf{occ}(e)])) \quad ◆$$

Fact 8.2.3.6

The following equation stating a connection between the specifications *SOURCE* and *NOPT* is valid:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$
$$regval(1, code^{NOPT}(\mathbf{occ}(sv[e])) = value(\mathbf{occ}(sv[e])) \quad ◆$$

Proof

Follows immediately from the fact:

$$\forall\, sv_{Expr \to Expr}.\ \forall\, e_{Expr}.$$

$$val(usedregb^{NOPT}(sv[\mathbf{occ}(e)])) = value(sv[\mathbf{occ}(e)])$$
$$\quad \wedge\ usedregb^{NOPT}(\mathbf{occ}(sv[e]) = 1 \qquad ◆$$

Fact 8.2.3.7

It holds:

$OPT \rightsquigarrow_{\mathrm{beh}} NOPT$ ◆

Proof

Follows immediately from the above facts. ◆

8.2.4 Inverse Attribution

In this section we show, how the inverse attribution can be calculated, which can be used to derive e.g. the source code from a given target program. The derivation of such an inverse attribution is applicable e.g.

- if the source code was deleted and the target code exists,
- testing on high level of abstraction, e.g. of the compilation process yields the correct optimized code corresponding to a non optimized one,
- catch word „reengineering“: constructing the source code to a given target code depending on the compiler specification.

Let us assume a compiler specification and an implementation of this compiler specification, i.e. an executable program. The correctness of the compilation with the compiler relative to the compiler specification can be shown, if the source program is deduced from the target program using the compiler specification.

Another example is having a prototypical compiler specification from source language *A* to target language *B*, using inverse compilation a compiler from *B* to *A* is obtained without extra costs.

We consider the following property over the specification *NOPT* as an example:

$$\exists\ e_{Expr}.\ code(\mathbf{occ}(e)) = seq(regIs(1, 2), seq(regIs(2, 3), ADD(1, 1, 2)))$$

Start-configuration:

$$(\{\ y = seq(regIs(1, 2), seq(regIs(2, 3), ADD(1, 1, 2)))\ \}, [\,], \{\ y = code(\mathbf{occ}(e))\ \}) \vdash$$
$$(\{\ y = seq(regIs(1, 2), seq(regIs(2, 3), ADD(1, 1, 2))),$$
$$y = seq(x1, seq(x2, ADD(x3, x3, x4)))\ \},$$
$$[\ e\ /\ comp(e1, add, e2)\ ],$$
$$\{\ ...\ \}$$

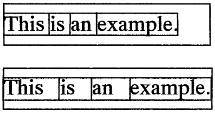
Leading to the starting point of the code generation, namely the term

$comp(natexpr(2), add, natexpr(3))$

obtained restricting the answer substitution to *e*.

8.3 Specifying Document Architecture Systems

The ideas of this subsection are based on [Bauer 94a, 94b]. The example is taken from a practical course in which the techniques of compiler construction (attribute grammars) were used in the framework of document architecture [SchreiberW 93]. One subtask of formatting a text is to jolt a list of boxes such that the sum of the sizes of the boxes is conform with a given size. An illustrative example is the computation of the justification of a line:



task: the sentence should fill the whole box.

desired result.

figure 37: justification example

This problem can be specified using the following loose attributed algebraic specification. *Hbox* defines a horizontal box with the constructor *mkhbox* taking the length to be obtained and a list of horizontal boxes *Hlist*. A *Hlist* is either an empty list (*mthlist*) or built by the constructor *mkhlist* with first argument being the local jolting factor, second argument the natural length and the rest of sort *Hlist*. The attribute *jlength* contains the computed size of the inner boxes which depends on the global jolting factor *jfactor* (in the next specification). *hboxlength* is the length which should be obtained by *jlength* of the boxes. *nlength* is the natural length of the boxes.

```
aspec LBOXES =
  enrich NAT by
    sorts Hbox, Hlist
    cons mkhbox: Nat, Hlist → Hbox,
           mthlist: → Hlist,
           mkhlist: Nat, Nat, Hlist → Hlist
    attrs synth   hboxlength, jlength, nlength: Hbox → Nat
    axioms for all i: Nat; l:Hlist.
    (1)     hboxlength(occ(mkhbox(i, l))) =
              ifnat eq_Nat(nlength(mkhbox(i, occ(l))), zero)
                then zero
                else jlength(mkhbox(i, occ(l))) fi
endspec
```

Another loose specification is:

```
aspec LBOXES2 =
  enrich NAT by
    sorts Hbox, Hlist
    cons mkhbox: Nat, Hlist → Hbox,
           mthlist: → Hlist,
           mkhlist: Nat, Nat, Hlist → Hlist
  attrs synth   hboxlength, jlength, nlength, ljfactor: Hbox → Nat
         inh      jfactor: Hbox → Nat
  axioms for all i, i1, i2: Nat; l: Hlist.
    (1)     jfactor(mkhbox(i, occ(hl))) * nlength(mkhbox(i, occ(hl))) =
              hboxlength(occ(mkhbox(i, hl))) - ljfactor(mkhbox(i, occ(hl))),
    (2)     nlength(sv[occ(mthlist)]) = zero,
    (3)     jlength(sv[occ(mthlist)]) = zero,
    (4)     ljfactor(sv[occ(mthlist)]) = zero,
    (5)     nlength(sv[occ(mkhlist(i1, i2, hl))]) = i2 + nlength(sv[mkhlist(i1, i2, occ(hl))]),
    (6)     jlength(mkhbox(i, sv[occ(mkhlist(i1, i2, hl))])) =
              i1 + i2 * jfactor(mkhbox(i, occ(sv[mkhlist(i1, i2, hl)]))) +
              jlength(mkhbox(i, sv[mkhlist(i1, i2, occ(hl))])),
    (7)     ljfactor(sv[occ(mkhlist(i1, i2, hl))]) = i1 + ljfactor(sv[mkhlist(i1, i2, occ(hl))])
endspec
```

The attribute *ljfactor* computes the sum of the first components of the *Hlist* elements being the local jolting factors. *nlength* is the sum of the second components, namely the sum of the natural lengthes of the inner boxes. The first component of *mkhbox* is the real size of the box, i.e. of the whole box from above.

For this specification several correct attributed trees are obtained since *LBOXES*2 is a loose specification, too. Possible attributions for the term

mkhbox(30, *mkhlist*(4, 2, *mthlist*)

are shown in the figure 38 and figure 39:

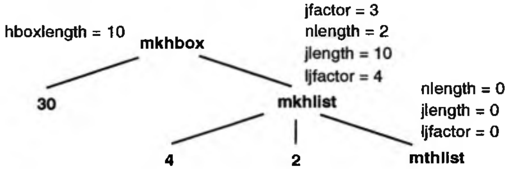


figure 38: several correct attributions - part 1

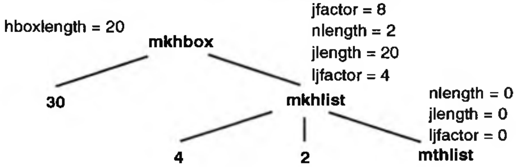


figure 39: several correct attributions - part 2

In order to obtain a single attribution the loose specification of the attribute *hboxlength* is fixed in the specification *BOXES* with axiom (1):

```
aspec BOXES =
  enrich NAT by
    sorts Hbox, Hlist
    cons mkhbox: Nat, Hlist → Hbox,
         mthlist: → Hlist,
         mkhlist: Nat, Nat, Hlist → Hlist
  attrs synth  hboxlength, jlength, nlength, ljfactor: Hbox → Nat
        inh    jfactor: Hbox → Nat
  axioms for all i, i1, i2: Nat; hl: Hbox; l: Hlist.=
    (1)  hboxlength(occ(mkhbox(i, hl))) = i,
    (2)  jfactor(mkhbox(i, occ(hl))) * nlength(mkhbox(i, occ(hl))) =
           hboxlength(occ(mkhbox(i, hl))) - ljfactor(mkhbox(i, occ(hl))),
    (3)  nlength(sv[occ(mthlist)]) = zero,
    (4)  jlength(sv[occ(mthlist)]) = zero,
    (5)  ljfactor(sv[occ(mthlist)]) = zero,
    (6)  nlength(sv[occ(mkhlist(i1, i2, hl))]) = i2 + nlength(sv[mkhlist(i1, i2, occ(hl))]),
    (7)  jlength(mkhbox(i, sv[occ(mkhlist(i1, i2, hl))])) =
           i1 + i2 * jfactor(mkhbox(i, occ(sv[mkhlist(i1, i2, hl)]))) +
           jlength(mkhbox(i, sv[mkhlist(i1, i2, occ(hl))])),
    (8)  ljfactor(sv[occ(mkhlist(i1, i2, hl))]) = i1 + ljfactor(sv[mkhlist(i1, i2, occ(hl))])
endspec
```

We prove the implementation of *LBOXES* by *BOXES*.

Therefore the following fact is necessary:

Fact 8.3.1

$\forall\, sv_{Hlist \rightarrow Hlist}.\ \forall\, x_{Hlist}.$

$$\begin{aligned}
&\mathit{jlength}(\mathit{mkhbox}(i, sv[\mathbf{occ}(x)])) - \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathbf{occ}(x)])) = \\
&\qquad \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[x]))) * \mathit{nlength}(\mathit{mkhbox}(i, sv[\mathbf{occ}(x)])).
\end{aligned}$$ ♦

Proof

The complete set being considered is:

$CS_x = \{\ sv_{Hlist \rightarrow Hlist}\ \}$

$CS_x = \{\ \mathit{mthlist}, \mathit{mkhlist}(i1, i2, l)\ \}$[21]

mthlist:

$$\begin{aligned}
&\mathit{lhs}\text{:}\ \ \mathit{jlength}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mthlist})])) - \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathit{occ}(\mathit{mthlist})])) = \\
&\qquad \mathit{zero} - \mathit{zero} = \mathit{zero} \\
&\mathit{rhs}\text{:}\ \ \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mthlist}]))) * \mathit{nlength}(\mathit{mkhbox}(i, sv[\mathit{occ}(\mathit{mthlist})])) = \\
&\qquad \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mthlist}]))) * \mathit{zero} = \mathit{zero}
\end{aligned}$$

$\mathit{mkhlist}(i1, i2, l)$:
induction assertion:

$$\begin{aligned}
&\mathit{jlength}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l))])) - \\
&\quad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l))])) = \\
&\quad \mathit{jfactor}(\mathit{mkhbox}(i, \mathit{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) * \\
&\quad \mathit{nlength}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l))]))
\end{aligned}$$

Thus we get:

$$\begin{aligned}
&\mathit{jlength}(\mathit{mkhbox}(i, sv[\mathit{occ}(\mathit{mkhlist}(i1, i2, l))])) - \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) = \\
&\quad i1 + i2 * \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) + \\
&\qquad \mathit{jlength}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathit{occ}(l))])) - \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) = \\
&\quad i1 + i2 * \mathit{jfactor}(\mathit{mkhbox}(i, \mathit{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) + \\
&\qquad \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) * \\
&\qquad \mathit{nlength}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathit{occ}(l))])) + \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l))])) - \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) = \\
&\quad i1 + i2 * \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l)]))) + \\
&\qquad \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) * \\
&\qquad (\mathit{nlength}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) - i2) + \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathit{mkhlist}(i1, i2, \mathbf{occ}(l))])) - \\
&\qquad \mathit{ljfactor}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) = \\
&\quad i1 + i2 * \mathit{jfactor}(\mathit{mkhbox}(i, \mathit{occ}(sv[\mathit{mkhlist}(i1, i2, l)]))) + \\
&\qquad \mathit{jfactor}(\mathit{mkhbox}(i, \mathbf{occ}(sv[\mathit{mkhlist}(i1, i2, l)])))) * \\
&\qquad (\mathit{nlength}(\mathit{mkhbox}(i, sv[\mathbf{occ}(\mathit{mkhlist}(i1, i2, l))])) - \mathrm{i2}) +
\end{aligned}$$

[21] Stating that the usual term induction is sufficient for the proof of the property.

(*ljfactor*(*mkhbox*(*i*, *sv*[**occ**(*mkhlist*(*i*1, *i*2, *l*))])) - i1) -
ljfactor(*mkhbox*(*i*, *sv*[**occ**(*mkhlist*(*i*1, *i*2, *l*))])) =
jfactor(*mkhbox*(*i*, **occ**(*sv*[*mkhlist*(*i*1, *i*2, *l*)]))) *
nlength(*mkhbox*(*i*, *sv*[**occ**(*mkhlist*(*i*1, *i*2, *l*))])) =
jfactor(*mkhbox*(*i*, **occ**(*sv*[*mkhlist*(*i*1, *i*2, *l*)]))) *
nlength(*mkhbox*(*i*, *sv*[**occ**(*mkhlist*(*i*1, *i*2, *l*))])) ◆

With this fact it is now possible to proof the implementation relation:

Fact 8.3.2

It holds:

LBOXES is implemented by *BOXES*. ◆

Proof

We have to show the validity of the axioms of *LBOXES* with the axioms of *BOXES*.

Axiom(1) can be shown with the fact from above:

jlength(*mkhbox*(*i*, **occ**(*l*))) =
jfactor(*mkhbox*(*i*, **occ**(*l*))) * *nlength*(*mkhbox*(*i*, **occ**(*l*))) +
ljfactor(*mkhbox*(*i*, **occ**(*l*))) =
(*hboxlength*(**occ**(*mkhbox*(*i*, *l*))) - *ljfactor*(*mkhbox*(*i*, **occ**(*l*)))) /
nlength(*mkhbox*(*i*, **occ**(*l*)))) * *nlength*(*mkhbox*(*i*, **occ**(*l*))) +
ljfactor(*mkhbox*(*i*, **occ**(*l*))) = *hboxlength*(**occ**(*mkhbox*(*i*, *l*))).

neglecting *ifnat.then.else.if.* ◆

For the specification *BOXES* we demonstrate how the attribute evaluation ordering, i.e. the visit sequence, can be determined. The starting point for determining the attribute evaluation ordering is the characteristic term being

mkhbox(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])

with its instantiated local dependency set:

{ { *hboxlength*(**occ**(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) },

{ *jfactor*(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
nlength(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
hboxlength(**occ**(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
ljfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])))) },

{ *nlength*(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) },

{ *jlength*(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) },

{ *ljfactor*(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) },

{ *nlength*(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])),
nlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) },

{ *jlength*(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])),
jfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
jlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) },

{ *ljfactor*(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])),
ljfactor(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) }}

This set is the basis for determining the attribute evaluation ordering.

The set of minimal elements *me* is

{ *hboxlength*(**occ**(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
nlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])),
jlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])),
ljfactor(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) }

resulting in:

me < *ljfactor*(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) <
ljfactor(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
ljfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])))) and

me < *nlength*(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) <
nlength(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
nlength(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])))

me < { *nlength*(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
hboxlength(**occ**(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),
ljfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])))) } <
jfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) and

me < { *jfactor*(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) ,
jlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) } <
jlength(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
jlength(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),

This partial ordering is changed to a total ordering:

hboxlength(**occ**(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) <
nlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) <
ljfactor(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) <
jlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, *sv*2[**occ**(*mthlist*)])])) <
nlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) <
ljfactor(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) <
nlength(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
ljfactor(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
nlength(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) <
ljfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])])))) <
jfactor(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))) <
jlength(*mkhbox*(*i*1, *sv*1[*mkhlist*(*i*2, *i*3, **occ**(*sv*2[*mthlist*]))])) <
jlength(*mkhbox*(*i*1, *sv*1[**occ**(*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*]))])) <
jlength(*mkhbox*(*i*1, **occ**(*sv*1[*mkhlist*(*i*2, *i*3, *sv*2[*mthlist*])]))),

with corresponding visit sequence:

hboxlength for nodes marked with *mkhbox*

nlength, *ljfactor*, (*jfactor*), *jlength* for nodes marked with *mthlist* and *mkhlist*. The global jolting factor (*jfactor*) is set in brackets because its value is calculated once.

BOXES can be implemented by *CBOXES* with directed attribute equations of the following form:

```
aspec CBOXES =
  enrich NAT by
    sorts Hbox, Hlist
    cons mkhbox: Nat, Hlist → Hbox,
           mthlist: → Hlist,
           mkhlist: Nat, Nat, Hlist → Hlist
    attrs synth  hboxlength, jlength, nlength, ljfactor: Hbox → Nat,
           inh     jfactor: Hbox → Nat
    axioms for all i, i1, i2: Nat; hl: Hbox; l:Hlist.
    (1)    nlength(occ(mkhbox(i, hl))) = nlength(mkhbox(i, occ(hl))),
    (2)    jlength(occ(mkhbox(i, hl))) = jlength(mkhbox(i, occ(hl))),
    (3)    hboxlength(occ(mkhbox(i, hl))) = i,
    (4)    jfactor(mkhbox(i, occ(hl))) =
             ifnat eqnat(nlength(mkhbox(i, occ(hl))), zero)
                   then zero
                   else (i-ljfactor(mkhbox(i, occ(hl)))) / nlength(mkhbox(i, occ(hl))) fi,
    (5)    nlength(sv[occ(mthlist)]) = zero,
    (6)    jlength(sv[occ(mthlist)]) = zero,
    (7)    ljfactor(sv[occ(mthlist)]) = zero,
    (8)    nlength(sv[occ(mkhlist(i1, i2, hl))]) = i2 + nlength(sv[mkhlist(i1, i2, occ(hl))]),
    (9)    jlength(sv[occ(mkhlist(i1, i2, hl))]) =
             i1 + i2 * jfactor(sv[occ(mkhlist(i1, i2, hl))]) + jlength(sv[mkhlist(i1, i2, occ(hl))]),
    (10)   ljfactor(sv[occ(mkhlist(i1, i2, hl))]) = i1 + ljfactor(sv[mkhlist(i1, i2, occ(hl))]),
    (11)   jfactor(sv[mkhlist(i1, i2, occ(hl))]) = jfactor(sv[occ(mkhlist(i1, i2, hl))])
endspec
```

Fact 8.3.3

It holds:

BOXES is implemented by *CBOXES*. ◆

Proof

We have to show the validity of the axioms of *BOXES* with the axioms of *CBOXES*.

The only interesting fact to show is that the remote access is handled using local attribute access (axiom (11)). ◆

Fact 8.3.4

It holds:

LBOXES is implemented by *CBOXES*. ◆

Proof

Follows immediately by the transitivity of the implementation relation. ◆

9 Further Research Directions

In this chapter some ideas are discussed for further research in the presented framework and several points of extensions are studied.

9.1 Prototypical Implementations

Up to now only some restricted implementations of the presented specification technique exist. The most elaborated implementation is a theorem prover for the presented proof principle of attributed term induction [Duschl 94; Weiß 95] calculating proof obligations which are shown by the TIP system [Fraus 94a, 94b]. This theorem prover has delivered some restrictions which have been overcome using complete sets and induction orderings. This system allows to show e.g. the properties of the ISDN telephone case study. Beyond the implementation of the attributed term induction the attributed signature flow analysis problem of reachability is implemented.

It is planned to perform a prototypical implementation of the other aspects, especially narrowing and attribute evaluation in the functional programming language *Gofer*. First steps are already taken. It exists a rudimental scanner, parser and unparser for the specifications and a first implementation of the unification calculus.

9.2 State-Based Rewriting

In usual attribute grammar systems scanners and parsers build from a text file the abstract syntax tree. The attribution is performed either while the syntax tree is constructed or after the syntax tree is built.

But e.g. in the framework of user interface specifications rules should be defined how interactions change the dialogue tree. A rule is applied to a tree, if the action on the user interface is fired. These rules are similar to the state-based rewrite rules applied in Maude (cf. e.g. [Meseguer 90, 93a, 93b]).

Example 9.2.1 (state-based rewriting)

An example showing the use of state-based rewrite-rules is the extension of the telephone case study of section 8.1 using rewrite-rules for the transformation of the dialogue tree.

Assuming the following (conditional) state-based rewrite-rules

(1) mtDialog → mkCall(CALL, x)
(2) mkCall(CALL, x) → mkCall(CALL, EnterTNumber(y))
(3) istelephoning(sv[**occ**(mkCall(x, y))]) = true ⇒
    mkCall(x, y) → mkTask(mkCall(x, y), END)

the following dialogue sequence is derivable (We assume that each rewrite rule is fired by a particular user interaction.):

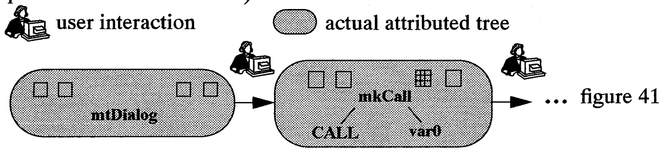


figure 40: dialogue sequence - part 1

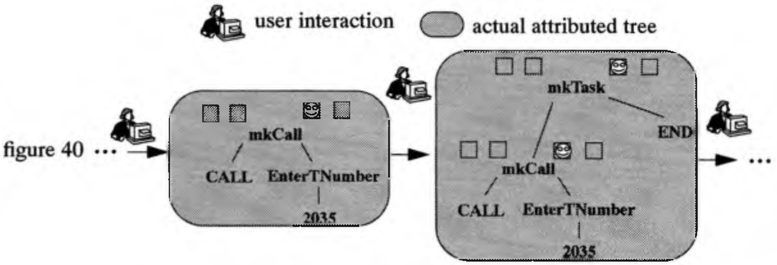


figure 41: dialogue sequence - part 2

The technique of state-based rewrite-rules can be used for the specification of the dynamic semantics of programming languages, especially for programming languages supporting parallel features. In this context the application of the rewrite rules is performed automatically and describes the trace of the program.

9.3 Communication between Different Processes and Conditional Equations

As already stated the extended term notion allows to specify attribute dependencies between different terms.

[Hirshfeld et al. 96] shows that processes can be described using context free grammars. The attribution between different terms (called inter-attribution) can be viewed as the communication between different processes.

An example for such an inter-attributed specification is the description of the communication of a producer and consumer process. The idea is to describe as well the trace of the producer process as the consumer process using grammars. Examples of traces are shown in the figure 42:

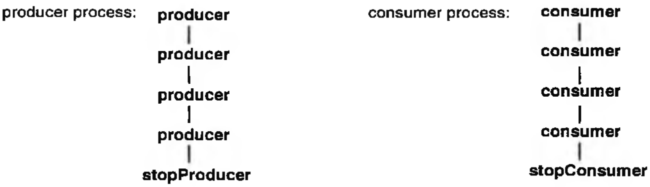


figure 42: producer and consumer process

Now the communication between both processes can be described by attribution. The producer process sends informations to the consumer process and the consumer process works up the datas. For the communication of both processes counters are introduced, one for the producer and one for the consumer process. Having no counters a non-deterministic information exchange between the two processes is described.

In attributed algebraic specifications with inter-attribute dependencies the notion of inherited attributes is extended to those attributes accessed in a different term.

An example for such an attributed algebraic specification is

```
aspec PRODUCER-CONSUMER =
  enrich NAT by
    sorts Producer, Consumer
    cons producer: Producer → Producer,
           consumer: Consumer → Consumer,
           stopProducer: → Producer,
           stopConsumer: → Consumer
    opns produce: Nat → Nat,
           consume: Nat → Nat
    attrs inh counterProducer, produceResult: Producer → Nat,
                counterConsumer, consumeResult: Consumer → Nat
    axioms for all  svP: Producer → Producer; svC: Consumer → Consumer;
                          p: Producer; c: Consumer.
    (1) counterProducer(occ(p)) = 0,
    (2) counterConsumer(occ(c)) = 0,
    (3) counterProducer(svP[producer(occ(p))]) = counterProducer(svP[occ(producer(p))]) + 1,
    (4) counterConsumer(svC[consumer(occ(c))]) =
          counterConsumer(svC[occ(consumer(c))]) + 1,

    (5) produceResult(svP[occ(p)]) = produce(counterProducer(svP[occ(p)])),
    (6) counterProducer(svP[occ(p)]) = counterConsumer(svC[occ(c)])
          ⇒ consumeResult(svC[occ(c)]) = consume(produceResult(svP[occ(p)]))
endspec
```

The attributed algebraic specification *PRODUCER-CONSUMER* describes the behaviour of two processes, one producer process (sort *Producer*) and one consumer process (sort *Consumer*). Furthermore, each produced value of the producer process is consumed by the consumer process. The synchronisation is performed with the *counterProducer* and *counterConsumer* attribute.

The initialization of the counters is performed in the axioms (1) and (2). Axioms (3) and (4) define the increment of the counters. Axiom (5) specifies the result of the produced value at a node of sort *Producer* depending on the actual counter of this node. If the counters of both processes are equal then the produced result of the producer process is consumed by the operation *consume* and stored in the attribute *consumeResult* of the consumer process (axiom (6)). The functions *produce* and *consume* are not specified, since arbitrary functions can be used. Note, that axiom (6) is a conditional equation to shorten notation, but we can also give an equational specification for it. But handling communication of processes in a practical way it is necessary to deal with conditional equations.

Given two terms

$$t_1 \equiv producer(producer(producer(producer(stopProducer))))$$

and

$$t_2 \equiv consumer(consumer(consumer(consumer(stopConsumer))))$$

the attribution looks like (with the producer function doubling the counter and the consumer function incrementing the delivered value):

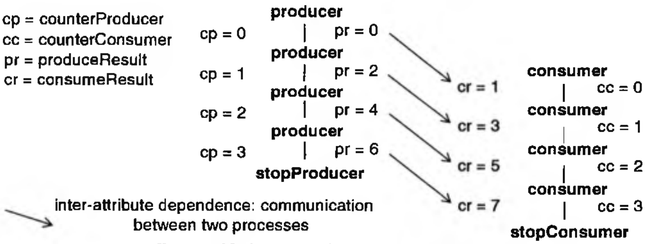


figure 43: inter-attribute dependencies

For the producer process the values of the counter and the produced result is calculated from the root to the tips, by incrementing the counter with each function and computing the produced result by doubling the counter. Analogous the counter of the consumer can be determined. Axiom (6) states that under the assumption that the counter of both traces are equal the consumed result of the consumer is the incremented produced result of the producer process. ◆

The idea of inter-attribute dependency analysis is studied in the following

For each process to be considered a characteristic term has to be constructed being in the example of the producer - consumer:

svC[*consumer*(*svC*2[*stopConsumer*])] and *svP*[*producer*(*svP*2[*stopProducer*])]

The attribute dependency between the both terms is visualized in figure 44:

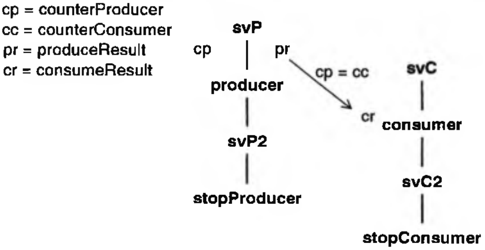


figure 44: inter-attribute example: producer - consumer

The result is a conditional attribute dependence relation between different processes, because of the conditional attribute equations:

counterConsumer(*svC*[**occ**(*consumer*(*svC*2[*stopConsumer*]))]) =
 counterProducer(*svP*[**occ**(*producer*(*svP*2[*stopProducer*]))])
 ⇒ *consumeResult*(*svC*[**occ**(*consumer*(*svC*2[*stopConsumer*]))]) =
 produceResult(*svP*[**occ**(*producer*(*svP*2[*stopProducer*]))])

Starting with the characteristic terms the ordering on the attribute dependencies can be calculated analogous to the proposed approach with the main difference that a conditional attribute evaluation ordering is obtained.

Conditional equations are also useful for the description of attribute dependencies within a tree. Among the extensions of the attribute dependency analysis and the attribute evaluation strategy, the calculi for the universally and existentially quantified formulae have to be adapted in the usual way to handle conditional equations.

9.4 User Interfaces

The FLUID system, whose theoretical foundations were presented here, is currently under development, whereby prototypes of BOSS and PLUG-IN already exist. The FUSE methodology and tools have been applied successfully to a number of examples (ISDN phone simulation, user interface for a literature research system, user interface for a home banking system, formula editor for LATEX). In the future the level of compatibility of the FUSE development environment to other model based methodologies and tools can be increased. E.g. for setting up the problem domain model, like OOA, BON and ERA data models in addition to the currently supported algebraic specification technique.

In order to gain more practical experience with the FUSE–methodology and the related tools, we plan to organize a course in user interface specification and generation at the Munich University of Technology.

9.5 Further Case Studies

Assuming a system to handle attributed algebraic specifications some complex case studies can be considered like a complete compiler for a small imperative or functional programming language. These case studies will detect shortcomings of the specification technique and will inspire ideas where the new formalism has to be adapted.

In the framework of user interface specification a complex example can be the specification of a part of the user interface of an SAP/R3 application or another business oriented application.

In the framework of document architecture or compiler construction the complete courses given at the Munich University of Technology can be used as a basis for a formal development of these software project.

Together with other discussed extensions the specification of the semantics of programming languages is possible. Investigations in this direction are desirable in combinations with specification and verification of compilers.

9.6 Attribute Evaluation and Calculi

A new attribute evaluation technique for undirected attribute equations was investigated. As in usual attribute grammar systems incremental attribute evaluation techniques (cf. e.g. [Reps et al. 83]) can be adapted to the new approach. Such an incremental attribute evaluation is desirable if a programming environment is generated from an attributed algebraic specification.

Moreover, for the specification of the semantics of programming languages or for optimizing compilers (e.g. performing dead-code elimination or liveness tests) cyclic attribute dependencies are necessary. Here the well known techniques from e.g. [Farrow 86; Jones 90] can be adapted to undirected attribute equations. The proof principles have to be extended to handle these cyclic dependencies introducing some rules for fixpoint induction.

Considering the global attribute dependencies and the underlying signature the global attribute dependencies can be automatically changed to local attribute dependencies based on the unification calculus.

9.7 „Specification Sugar“

As seen in the related work existing attribute grammar systems allow the use of „specification sugar“ to obtain specifications which are more readable and can easily be changed. One possibility for extending the new approach is the use of order-sorted signatures. This extension subsumes the object-oriented inheritance and simplifies notation.

The other notations presented in the related work section (section 7.1) can be put on top of the new specification technique.

10 Conclusions

We have investigated a new specification framework combining and extending the specification formalism of attribute grammars and algebraic specifications. Algebraic specifications are extended to describe context dependent information in an intuitive way and showing their correctness.

The new specifications technique can be applied in a formal software engineering process starting with an abstract specification and arriving at a usual attribute grammar after several implementation steps. The correctness of the software can be guaranteed if the correctness of each refinement step is shown. For this purpose notions of implementation relations are introduced and proof theoretical characterizations are given. Especially the new notion of behaviour - based on an intuitive idea - has been proven to be a good abstraction mechanism for attributed algebraic specifications. To start with specifications where several design decisions are left open undirected attribute equations are an essential component of the new technique.

The presented structuring mechanisms increase the re-use of specifications and allow the handling of complex software projects. E.g. in the framework of compiler construction the specification for the problems of identification, typing or code optimizations can be proved correct and can be put in libraries. Since the implementation relations are monotone wrt. the specification building operations such a proceeding is supported. Moreover, well known extensions of attribute grammars are subsumed in the proposed approach. Advantages of other techniques are contained in attributed algebraic specifications neglecting sometimes their shortcomings.

The presented calculi can be used for the verification and the prototypical use of the specifications. Using the specifications in a prototypical way efficient attribute evaluation strategies were developed. The calculated visit sequence can be used as an input for usual systems generating attribute evaluators.

Each case study shows some interesting aspects of the considered new specification technique. Especially the non trivial compiler example of register placing and its optimizations shows the usability of the calculi, proof principles and implementation relations.

The presented approach combines the advantages of attribute grammars:
- intuitivity, since attributes are associated with nodes of a tree;
- efficiency, since an efficient attribute evaluation algorithm for undirected attribute equations is presented;
- context dependent information, since synthesized and inherited attributes are supported and
- distinction of syntax and semantics specification, since the syntax is defined by constructor terms and their semantics using attribution;

with those of algebraic specifications:
- precise model class semantics, since a model class and behavioural class semantics is given for the new technique;
- theorem-proving techniques, since calculi are investigated for existentially and universally quantified formulae;
- deductive aspects, since the calculus for existentially formulae can be used to derive

parts of e.g. a program or a dialogue;
- implementation relations, since implementation relations are defined for the observable and non-observable case and
- abstraction, since an intuitive notion of behaviour was investigated.

All these aspects are well known in one of the two approaches, but they have not been considered in the other one. No formalism can be found in the literature supporting these aspects in common. The main contribution of this thesis is that a technique was investigated subsuming all these aspects which are necessary in a formal software engineering process.

Overall a specification technique was developed combining two well accepted approaches with their advantages. It was shown how attributed algebraic specifications can be used in a formal software engineering process. The necessity using formal methods are illustrated by the error-prone software in the market. Combining formal methods with generation techniques decrease the amount of error, since on the one side the input of the generation process can be proven correct and on the other side generating programs is less error-prone than programming software by hand. Proving the correctness of the generator an error free program is obtained from its high-level specification. In the new approach a formal development process as well as generation are supported.

Table of Figures

Table of Notations and Abbreviations

The following notations and abbreviations are used:

$\equiv$	denotes the syntactical equality
$\not\equiv$	denotes the syntactical inequality
$\lvert S\rvert$	denotes the cardinality of a set S
$dom(f)$	denotes the domain of a function f
$codom(f)$	denotes the codomain of a function f
S^*	denotes the set of all words over S including the empty word ε
S^+	denotes the set of all words over S without the empty word ε
π_i	denotes the projection to the i-th component of a tuple
2^S	denotes the powerset of S
beh.	is the abbreviation for *behaviour* or *behavioural*
c.f.	is the abbreviation for *confer*
ed.	is the abbreviation for *editor*
eds.	is the abbreviation for *editors*
e.g.	is the abbreviation for *for example*
i.e.	is the abbreviation for *id est*
iff	is the abbreviation for *if and only if*
LNCS	is the abbreviation for *Lecture Notes in Computer Science*
pp.	is the abbreviation for *pages*
Proc.	is the abbreviation for *Proceedings*
p.r.s.	is the abbreviation for *primitive recursive scheme*
s.t.	is the abbreviation for *such that*
wlog.	is the abbreviation for *without loss of generality*
wrt.	is the abbreviation for *with respect to*

References

[Ada 83] *Ada Programming Language,* Military Standard ANSI/MIL-STD-1815A, 1983

[ADJ 76] J. A. Goguen, J. W. Thatcher, E. G. Wagner: *An initial algebra approach to the specification, correctness and implementation of abstract data types,* IBM research report RC 6487 (1976), also in: Current Trends in Programming Methodology, (ed.) R. T. Yeh, *4,* Data Structuring, Prentice Hall, pp. 80-149, 1978

[Aho et al. 86] A. V. Aho, R. Sethi, J. D. Ullman: *Compilers, Principles, Techniques and Tools,* Bell Telephone Laboratories, Inc. Reading, Mass., 1986

[Alblas 89] H. Alblas: *Iteration of Transformation Passes over Attributed Program Trees,* in: Acta Informatica, *27,* pp. 1-40, 1989

[Alblas, Melichar 90] H. Alblas, B. Melichar: *Attribute Grammars: Applications and Systems,* in: LNCS *545,* Springer, Berlin, 1990

[Balzert 93] H. Balzert: *Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur,* in: Softwaretechnik, *93(13),* 1993

[Balzert 94] H. Balzert: *Das JANUS-System,* in: Informatik Forschung und Entwicklung, 1994

[Balzert 95] H. Balzert: *From OOA To GUI - The Janus System,* in: Proc. of Interact 95, IFIP, *7,* 1995

[Bauer 94a] B. Bauer: *Attributed Term Induction - A Proof Principle for Attribute Grammars,* technical report, TUM-I9403, Institut für Informatik, Technische Universität München, February 1994

[Bauer 94b] B. Bauer: *Proving Properties over Attribute Grammars,* in: Workshop der GI-Fachgruppe 2.1.3, Semantikgestützte Analyse, Entwicklung und Generierung von Programmen, (eds.) U. Meyer, G. Snelting, technical report, 9402, AG Informatik, Justus-Liebig-Universität Giessen, Schloß Rauischholzhausen, pp. 119-130, March 1994

[Bauer 95] B. Bauer: *Proving the Correctness of Formal User Interface Specifications,* in: Proc. of the 2nd Eurographics Workshop on Design, Specification, Verification of Interactive Systems, Chateau da Bonas, Frankreich, June 6-9, Springer Eurographics Series, Springer, Berlin, pp. 224-241, 1995

[Bauer 96] B. Bauer: *Generating User Interfaces from Formal Specifications of the Application,* in: Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96, Namur, 5-7 June 1996, J. Vanderdonckt (ed.), Presses Universitaires de Namur, Namur, 1996

[Bauer, Hennicker 93] B. Bauer, R. Hennicker: *Proving the Correctness of Algebraic Implementations by the ISAR System,* in: Proc. of the International Symposium on Design and Implementation of Symbolic Computation Systems 93, LNCS *722,* Springer, Berlin, pp. 3-16, 1993

[Bauer, Hennicker 94] B. Bauer, R. Hennicker: *Behavioural Program Development with the ISAR System,* in: Proc. Workshop Systems for Computer-Aided Specification, Development and Verification, (eds.:) B. Buth, R. Berghammer, technical report, 9416, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, pp. 140-153, 1994

[Berghammer et al. 87] R. Berghammer, H. Ehler, H. Zierer: *Towards an algebraic specification of code generation,* technical report, TUM-I8707, Institut für Informatik, Technische Universität München, June 1987

[Bergstra et al. 89] J. A. Bergstra, J. Heering, P. Klint: *Algebraic Specifications,* ACM Press Frontier Series, Addison-Wesley, New York, 1989

[Bernot, Bidoit 91] G. Bernot, M. Bidoit: *Proving the correctness of algebraically specified software: Modularity and Observability issues,* in: Proc. of the AMAST Conference 91, Workshop in Computing Series, Springer, Berlin, pp. 216-242, 1991

[Bidoit et al. 94] M. Bidoit, R. Hennicker, M. Wirsing: *Characterizing behavioural semantics and abstractor semantics*, in: Proc. ESOP '94, 4th European Symposium on Programming, LNCS *788*, Springer, Berlin, pp. 105-119, 1994

[Bidoit et al. 95] M. Bidoit, R. Hennicker, M. Wirsing: *Behavioural and abstractor specifications*, in: Science of Computer Programming, *25* (*2-3*), pp. 149-186, 1995

[Bidoit, Hennicker 94] M. Bidoit, R. Hennicker: *Proving Behavioural Theorems with Standard First-Order Logic*, in: Proc. Algebraic and Logic Programming, 4th International Conference, ALP 94, LNCS *850*, Springer, Berlin, pp. 41-58, 1994

[Bidoit, Hennicker 95] M. Bidoit, R. Hennicker: *Proving the Correctness of Behavioural Implementations*, in: Proc. AMAST 95, 4th International Conference on Algebraic Methodology and Software Technology, LNCS *906*, Springer, Berlin, pp. 152-168, 1995

[Bodart et al. 94] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, J. M. Vanderdonckt: *A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype*, in: Proc. of the Eurographics Workshop on Design, Specification and Verification of Interactive Systems, pp. 25-39, 1994

[Breu 89] R. Breu: *A Normal Form for Structured Algebraic Specifications*, technical report, MIP-8917, Fakultät für Mathematik und Informatik, Universität Passau, 1989

[Breu 91] R. Breu: *Algebraic Specification Techniques in Object Oriented Programming Environments*, LNCS *562*, Springer, Berlin, 1991

[Burstall, Goguen 77] R. M. Burstall, J. A. Goguen: *Putting theories together to make specifications*, in: Proc. 5th International Joint Conference on Artificial Intelligence, pp. 1045-1058, 1977

[Burstall, Goguen 80] R. M. Burstall, J. A. Goguen: *The semantics of CLEAR, a specification language*, in: Proc. Advanced Course on Abstract Software Specification, LNCS *86*, Springer, Berlin, pp. 292-332, 1980

[Chirica, Martin 76] L. M. Chirica, D. F. Martin: *An Algebraic Formulation of Knuthian Semantics*, in: Proc. of 17th Annual Symposium on Foundations of Computer Science, IEEE, Houston, Texas, 1976

[Chirica, Martin 79] L. M. Chirica, D. F. Martin: *An Order-Algebraic Definition of Knuthian Semantics*, in: Math. Systems Theory, *13*, pp. 1-27, 1979

[Courcelle, Deransart 88] B. Courcelle, P. Deransart: *Proofs of Partial Correctness for Attribute Grammars with Application to Recursive Procedures and Logic Programming*, in: Information and Computation, *78*, pp. 1-55, 1988

[Courcelle, Franchi-Zannettacci 82] B. Courcelle, P. Franchi-Zannettacci: *Attribute Grammars and Recursive Program Schemes I, II*, in: Theoretical Computer Science, *17*, North-Holland, pp. 163-191 and pp. 235-257, 1982

[Deransart et al. 88] P. Deransart, M. Jourdan, B. Lorho: *Attribute Grammars: Definitions, Systems and Bibliography*, LNCS *323*, Springer, Berlin, 1988

[Deursen 94] A. van Deursen: *Origin tracking in primitive recursive schemes*, report CS-R9401, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994, an earlier version appeared in: Proc. CSN '93, (ed.) H. A. Wijshof, pp. 132-143, 1993

[Dueck, Gormack 90] G. D. P. Dueck, G. V. Gormack: *Modular Attribute Grammars*, in: The Computer Journal, *33*, pp. 164-172, 1990

[Duschl 94] S. Duschl: *Ein interaktives System für Beweise über attributierten algebraischen Spezifikationen*, diploma thesis, Institut für Informatik, Technische Universität München, 1994

[Ehrich et al. 89] H.-D. Ehrich, M. Gogolla, U. W. Lipeck: *Algebraische Spezifikation abstrakter Datentypen*, B.G. Teubner, Stuttgart, 1989

[Ehrig, Mahr 85] H. Ehrig, B. Mahr: *Fundamentals of algebraic specifications* 1, EATCS Monographs on Theoretical Computer Science, *6*, Springer, Berlin, 1985

[Eickel 90] J. Eickel: *Logical and Layout Structures of Documents*, in: Computer Physics Communication, *61*, pp. 201-208, 1990

[Farrow 86] R. Farrow: *Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars*, in: Proc. of the SIGPLAN 86 Symposium on Compiler Construction, pp. 85-98, 1986

[Farrow et al. 92] R. Farrow, T. J. Marlowe, D. M. Yellin: *Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation*, in: Proc. of ACM Symposium on Principles of Programming Languages, 1992

[Foley et al. 91] J. D. Foley, W. C. Kim, S. Kovacevic, K. Murray: *UIDE - An Intelligent User Interface Design Environment*, in: Intelligent User Interfaces, (eds.) J. W. Sullivan, S. W. Tyler, Addison Wesley, ACM Press, pp. 339-384, 1991

[Foley et al. 93] J. D. Foley, P. N. Sukaviriya, T. Griffith: *A Second Generation User Interface Design Environment: The Model and the Runtime Architecture*, in: Proc. ACM INTERCHI 93, Conference on Human Factors in Computing Systems, Model-Based User Interface Development System 93, pp. 375-382, 1993

[Foley 95] J. D. Foley: *History, Resums and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation*, in: Proc. DSV-IS 94 Workshop, Springer, Berlin, 1995

[Fraus 94a] U. Fraus: *Inductive Theorem Proving for Algebraic Specifications - TIP System User's Manual*, technical report, MIP-9401, Fakultät für Mathematik und Informatik, Universität Passau, 1994

[Fraus 94b] U. Fraus: *Mechanizing Inductive Theorem Proving in Conditional Theories*, Ph. D. thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1994

[Futatsugi et al. 85] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, J. Meseguer: *Principles of OBJ2*, in: Proc. POPL, pp. 52-66, 1985

[Ganzinger 78] H. Ganzinger: *Optimierende Erzeugung von Übersetzerteilen aus implementierungsorientierten Sprachbeschreibungen*, Ph. D. thesis, Institut für Mathematik und Informatik, Technische Universität München, 1978

[Ganzinger, Giegerich 84] H. Ganzinger, R. Giegerich: *Attribute Coupled Grammars*, in: ACM SIGPLAN Notices, *19*(*6*), pp. 157-170, 1984

[Goguen, Meseguer 82] J. A. Goguen, J. Meseguer: *Universal realization, persistent interconnection and implementation of abstract modules*, in: Proc. ICALP '82, 9th Colloquium on Automata, Languages and Programming, (eds.) M. Nielsen, E. M. Schmidt, Aarhus, July 1982, LNCS *140*, Springer, Berlin, pp. 265-281, 1982

[Grey et al. 92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, W. M. Waite: *ELI: A Complete Flexible Compiler Construction System*, in: Communication of the ACM, *35*, pp. 121-131, 1992

[Grosch 89] J. Grosch: *AG - An Attribute Evaluator Generator*, technical report 16, GMD Forschungstelle Karlsruhe, 1989

[Guttag 75] J. V. Guttag: *The specification and application to programming of abstract data types*, Ph. D. thesis, University of Toronto, Dept. of Computer Science, also as technical report CSRG-59, 1975

[Hedin 89] G. Hedin: *An Object-Oriented Notation for Attribute Grammars*, in: Proc. ECOOP 89, pp. 329-346, 1989

[Hedin 92] G. Hedin: *Incremental semantic analysis*, Ph. D. thesis, Lund University, Sweden, 1992

[Hedin 94] G. Hedin: *An Overview of Door Attribute Grammars*, in: Proc. 5th International Conference on Compiler Construction, (ed.) Peter A. Fritzson, Edinburgh, LNCS *786*, Springer, Berlin, pp. 31-51, April 1994

[Heering et al. 94] J. Heering, K. Meinke, B. Möller, T. Nipkow (eds.): *HOA 93: An International Workshop on Higher-Order Algebra, Logic and Term Rewriting*, LNCS

816, Springer, Berlin, 1994

[Heering, Klint 89] J. Heering, P. Klint: *The syntax definition formalism SDF*, in: [Bergstra et al. 89], 1989

[Hennicker 88] R. Hennicker: *Beobachtungsorientierte Spezifikationen*, Ph. D. thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1988

[Hennicker 91] R. Hennicker: *Context induction: a proof principle for behavioural abstractions and algebraic implementations*, in: Formal Aspects of Computing, *3*(*4*), pp. 326-345, 1991.

[Hennicker 92] R. Hennicker: *A Semi-Algorithm for Algebraic Implementation Proofs*, in: Theoretical Computer Science, *104*, pp. 53-87, 1992

[Hennicker, Schmitz 96] R. Hennicker, C. Schmitz: *Object-oriented implementation of abstract data type specifications*, in: Proc. AMAST 96, 5th International Conference on Algebraic Methodology and Software Technology, Munich, July 1996. To appear in: Lecture Notes in Computer Science, 1996.

[Hennicker, Wirsing 93] R. Hennicker, M. Wirsing: *Behavioural specifications*, Working Material for the lectures of Marting Wirsing, Marktoberdorf, Institut für Informatik, Technische Universität München, 1993

[Herbrand 30] J. Herbrand: *Recherches sur la théorie de la démonstration*, in: J. Herbrand, Logical Writings, (ed.) W. Goldfarb, Harvard University Press, 1930

[Hirshfeld et al. 96] Y. Hirshfeld, M. Jerrum, F. Moller: *A polynomial-time algorithm for deciding bisimulation equivalence of normed context-free processes*, in: Journal of Theoretical Computer Science, *158*, April 1996

[Hofbauer, Kutsche 89] D. Hofbauer, R.-D. Kutsche: *Grundlagen des maschinellen Beweisens*, Vieweg, Braunschweig, Wiesbaden, 1989

[Hoppe 88] H. U. Hoppe: *Task-Oriented Parsing - A Diagnostic Method to be Used by Adaptive Systems*, in: Proc. of ACM CHI'88 Conference on Human Factors in Computing Systems, pp. 241-247, 1988

[Janssen et al. 93] Ch. Janssen, A. Weisbecker, J. Ziegler: *Generating User Interfaces from Data Models and Dialogue Net Specifications*, in: Proc. ACM INTERCHI 93, Conference on Human Factors in Computing Systems, Automated User Interface Generation, pp. 418-423, 1993

[Jones 90] L. G. Jones: *Efficient Evaluation of Circular Attribute Grammars*, in: ACM Transactions on Programming Languages and Systems, *12*(*3*), pp. 429-462, July 1990

[Kastens 91] U. Kastens: *Implementation of Visit-Oriented Attribute Evaluators*, in: Proc. of the International Summer School on Attribute Grammars, Application and Systems, LNCS *545*, Springer, Berlin, pp. 114-139, 1991

[Kastens, Waite 92] U. Kastens und W. M. Waite: *Modularity and Reusability in Attribute Grammars*, technical report, Reihe Informatik tr-ri-92-102, Universität-GH Paderborn, Fachbereich Mathematik-Informatik, July 1992, also in: Acta Informatica, *31*, pp. 601-627, 1994

[Katayama, Hoshino 81] T. Katayama, Y. Hoshino: *Verification of Attribute Grammars*, in: Proc. 8th POPL, Williamsburg, VA, pp. 177-186, January 1981

[Kernighan, Ritchie 78] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978

[Klint 93] P. Klint: *A meta-environment for generating programming environments*, in: ACM Transactions on Software Engineering and Methodology, *2*(*2*), pp. 176-201, 1993

[Knapik 91] T. Knapik: *Specifications with observable formulae and observational satisfaction relation*, in: Recent Trends in Data Type Specification, (eds.) M. Bidoit, C. Choppy, LNCS *655*, Springer, Berlin, pp. 271-291, 1991

[Knopp 90] J. W. Knopp: *Deklarativ erweiterte Attributgrammatiken: Funktionale Semantik und Implementierungskonzepte*, technical report, Institut für Informatik, TUM-I9030, Technische Universität München, 1990

[Knuth 68] D. E. Knuth: *Semantics of context-free languages*, in: Mathematical Systems Theory, *2*, pp. 127-145, 1968

[Kosiuczenko, Meinke 96] P. Kosiuczenko, K. Meinke: *On the Power of Higher-Order Algebraic Specification Methods*, in: Information and Computation, *124*, pp. 85-101, 1996

[Krönert et al. 89] G. Krönert, G. Lauber, H.-G. Mannes: *Spezifikation, Prototyping und Implementierung von interaktiven Systemen und Verwendung von attributierten Grammatiken*, in: Software-Entwicklung, pp. 225-238, 1989

[Liebl et al. 90] A. Liebl, J. Knopp, A. Poetzsch-Heffter: *Delayed evaluation: Making functional attribute grammars directly executable*, technical report, Institut für Informatik, Technische Universität München, 1990

[Liskov, Zilles 74] B. H. Liskov, S. N. Zilles: *Programming with abstract data types*. in: ACM SIGPLAN Notices, *9*, pp. 50-59, 1974

[Lonczewski 95] F. Lonczewski: *Using a WWW browser as an Alternative User Interface for Interactive Applications*, in: Poster Proceedings of the 3rd World Wide Web Conference, Roland Holzapfel (ed), Fraunhofer Institute for Computer Graphics, Darmstadt, Germany, 1995

[Lonczewski, Schreiber 96] F. Lonczewski, S. Schreiber: *The FUSE-System: an Integrated User Interface Design Environment*, in: Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces CADUI'96, Namur, 5-7 June 1996, J. Vanderdonckt (ed.), Presses Universitaires de Namur, Namur, 1996

[Magnusson et al. 90] B. Magnusson, M. Bengtsson, L. O. Dahlin, G. Fries, A. Gustavsson, G. Hedin, S. Minor, D. Oscarsson, M. Taube: *An Overview of the Mjolner/Orm Environment: Incremental Language and Software Development*, technical report, LU-CS-TR:90:57, Lund University, 1990

[Martelli, Montanari 82] A. Martelli, U. Montanari: *An efficicient unification algorithm*, in: ACM Transactions on Programming Languages Systems, *4(2)*, pp. 258-282, 1982

[McCarthy, Painter 67] J. McCarthy, J. Painter: *Correctness of a Compiler for Arithmetic Expressions*, in: Mathematical aspects of computer science, (ed.) J. T. Schwartz, Proc. of symposia in applied mathematics, *19*, pp. 33-41, 1967

[Meulen 94] E. A. van der Meulen: *Incremental Rewriting*, Ph. D. thesis, University of Amsterdam, 1994

[Meseguer 90] J. Meseguer: *A logical theory of concurrent objects*, in: ACM SIGPLAN Notices, *25(10)*, Proc. OOPSLA/ECOOP '90, pp. 101-115, Oktober 1990

[Meseguer 93a] J. Meseguer: *A logical theory of concurrent objects and its realization in the Maude language*, in: Research Directions in Object-Based Concurrency, (eds.) G. Agha, P. Wegner, A. Yonezawa, MIT Press, 1993

[Meseguer 93b] J. Meseguer: *Solving the inheritance anomaly on concurrent object-oriented programming*, in: ECOOP '93, Object Oriented Programming, (ed.) O. Nierstrasz, LNCS *707*, Springer, Berlin, pp. 220-246, 1993

[Meseguer, Winkler 92] J. Meseguer, T. Winkler: *Parallel programming in Maude*, in: Research Directions in High-level Parallel Programming Languages, (eds.) J.-P. Banâtre, D. Le Métayer, LNCS *574*, Springer Verlag, pp. 253-293, 1992

[Möller 87] B. Möller: *Higher-Order Algebraic Specifications*, Habilitationsschrift, Fakultät für Mathematik und Informatik, Technische Universität München, 1987

[Möncke, Wilhelm 91] U. Möncke, R. Wilhelm: *Grammar Flow Analysis*, in: Proc. Attribute Grammars, Application and Systems, International Summer School SAGA, LNCS *545*, Springer, Berlin, 1991

[Moore 56] E. F. Moore: *Gedanken-experiments on sequential machines*, in: Automata Studies, (eds.) C. E. Shannon, J. McCarthy, Princeton University Press, pp. 129-153, 1956

[Myers 88] B. A. Myers: *Tools for Creating User Interfaces: An Introduction and Survey*, technical report, CMU-CS-88-107, Carnegie Mellon University, Computer Science, 1988

[Nivela, Orejas 88] Mª. P. Nivela, F. Orejas: *Initial Behaviour Semantics for Algebraic Specifications*, LNCS *332*, Springer, Berlin, pp. 184-207, 1988

[Orejas et al. 91] F. Orejas, M. Navarro, A. Sanchez: *Implementation and behavioural equivalence: a survey*, in: Recent Trends in Data Type Specification, (eds.) M. Bidoit, C. Choppy, LNCS *655*, Springer, Berlin, pp. 93-125, 1991

[Padawitz, Wirsing 84] P. Padawitz, M. Wirsing: *Completeness of many-sorted equational logic revisited*, in: Bull. EATCS, *24*, pp. 88-94, 1984

[Payne 85] S. J. Payne: *Task-Action-Grammars*, in: Proc. INTERACT 84 Human-Computer Interaction, North-Holland, pp. 527-532, 1985

[Poetzsch-Heffter 91a] A. Poetzsch-Heffter: *Logic-Based Specification of Visibility Rule*, in: Programming Language Implementation and Logic Programming, (eds.) J. Maluszynski, M. Wirsing, LNCS *528*, Springer, Berlin, 1991

[Poetzsch-Heffter 91b] A. Poetzsch-Heffter: *Formale Spezifikation der kontextabhängigen Syntax von Programmiersprachen*, technical report, TUM-I9141, Ph. D. thesis, Institut für Informatik, Technische Universität München, 1991

[Poetzsch-Heffter 94] A. Poetzsch-Heffter: *Developing Efficient Interpreters Based on Formal Language Specifications*, in: Proceedings of the International Conference on Compiler Construction, Edinburgh, April 1994

[Poetzsch-Heffter 96] A. Poetzsch-Heffter: *Prototyping Realistic Programming Languages Based on Formal Specifications*, in: Acta Informatica, to appear, 1996

[Reade 89] Ch. Reade: *Elements of Functional Programming*, Addison-Wesley, 1989

[Reichel 81] H. Reichel: *Behavioural Equivalence - A Unifying Concept for Initial and Final Specification Methods*, in: Proc. Third Hung. Computer Science Conference, pp. 27-39, 1981

[Reisner 81] P. Reisner: *Formal Grammar and Human Factors Design of an Interactive Graphics System*, IEEE Transactions on Software Engineering, *7(2)*, March 1981

[Reps et al. 83] Th. Reps, T. Teitelbaum, A. Demers: *Incremental Context-Dependent Analysis of Language-Based Editors*, ACM Transactions on Programming Languages and Systems, *5(3)*, pp. 449-477, July 1983

[Reps, Teitelbaum 84] Th. Reps, T. Teitelbaum: *The Synthesizer Generator*, in: Proc. of the ACM SIGSOFT-SIGPLAN Software Engineering Symposium on Practical Software Development Environment, ACM SIGPLAN Notices, *19(5)*, pp. 42-48, May 84

[Robinson 65] J. A. Robinson: *A machine oriented logic based on the resolution principle*, Journal of the ACM, *12(1)*, pp. 23-41, 1965

[Sannella, Tarlecki 85] D. Sannella, A. Tarlecki: *On Observational Equivalence and Algebraic Specification*, LNCS *185*, Springer, Berlin, pp. 308-322, 1985

[Sannella, Tarlecki 88] D. Sannella, A. Tarlecki: *Towards formal development of programs from algebraic specifications: implementations revisited*, in: Acta Informatica, *25*, pp. 233-281, 1988

[Sannella, Wirsing 83] D. Sannella, M. Wirsing: *A kernel language for algebraic specifications and implementations*, in: Colloquium on Foundations of Computation Theory, (ed.) M. Karpinski, LNCS *158*, Springer, Berlin, pp. 413-427, 1983

[Schreiber 94a] S. Schreiber: *Specification and Generation of User Interfaces with the BOSS System*, in: Proc. East-West International Conference on Human-Computer Interaction EWHCI '94, St. Petersburg, Russia, August 2-6, 1994, (ed.) J. Gornostaev et al., Moskau, ICSTI, 1994, also in: Human Computer Interaction, Selected Papers EWHCI'94 Conference, LNCS *876*, Springer, Berlin, 1994

[Schreiber 94b] S. Schreiber: *The BOSS-System: Coupling Visual Programming with Model Based Interface Design*, in: Proc. Eurographics Workshop Design, Specification, Verification of Interactive Systems, (ed.) F. Paterno, Carrara, Italy, June 8-10, 1994, Springer Focus on Computer Graphics Series, 1994

[Schreiber 96] S. Schreiber: *Spezifikationstechniken und Generierungswerkzeuge für graphische Benutzungsoberflächen*, Ph. D. thesis, Institut für Informatik, Technische Universität München, to appear, 1996

[SchreiberW 93] W. Schreiber: *documentation to a compiler construction course*, Institut für Informatik, Technische Universität München, 1993

[SchreiberW 96] W. Schreiber: *Generierung von Dokumentverarbeitungssystemen aus Spezifikationen von Dokumentarchitekturen*, Ph. D. thesis, Institut für Informatik, Technische Universität München, to appear, 1996

[Scott, Strachey 71] D. Scott, C. Strachey: *Toward a Mathematical Semantics for Computer Languages*, in: Proc. of the Symposium on Computers and Automata, (ed.) J. Fox, Microwave Research Institute Symposia Series, Polytechnic Institute of Brooklyn, pp. 19-46, 1971

[Stoy 77] J. E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, 1977

[Swierstra, Vogt 91] D. Swierstra, H. Vogt: *Higher-Order Attribute Grammars*, in: Attribute Grammars, Applications and Systems, (eds.) H. Alblas, B. Melichar, LNCS *545*, Springer, Berlin, pp. 256-296, 1991

[Tauber 90] M. J. Tauber: *ETAG: Extended Task Action Grammars - A Language for the Description of the User's Task Language*, in: Proc. INTERACT 90 Human-Computer Interaction, North-Holland, pp. 163-168, 1990

[Thatcher et al. 77] J. W. Thatcher, J. A. Goguen, E. G. Wagner, J. B. Wright: *Initial Algebra Semantics and Continuous Algebras*, in: Journal of the Association of Computing Machinery, *24(1)*, pp. 68-95, January 1977

[Thatcher et al. 80] J. W. Thatcher, E. G. Wagner, J. B. Wright: *More on advice on structuring compilers and proving them correct*, in: Semantics-Directed Compiler Generation, (ed.:) Neil D. Jones, LNCS *94*, Springer, Berlin, pp. 165-188, 1980

[Vogt et al. 89] H. H. Vogt, S. D. Swierstra, M. F. Kuiper: *Higher-Order Attribute Grammars*, in: ACM SIGPLAN 89, Conference on Programming Language Design and Implementation, pp. 131-145, 1989

[Walz, Johnson 95] J. A. Walz, G. F. Johnson: *Inductive attribute grammars - A basis for incremental program execution*, in: Acta Informatica, *32*, pp. 117-144, 1995

[Weiß 95] M. Weiß: *Erweiterung eines interaktiven Systems für Beweise über algebraisch attributierten Grammatiken um Kontextinduktion*, documentation to a practical course in software development, Institut für Informatik, Technische Universität München, 1995

[Wilhelm, Maurer 92] R. Wilhelm, D. Maurer: *Übersetzerbau: Theorie, Konstruktion, Generierung*, Springer, Berlin, 1992

[Wirsing 82] M. Wirsing: *Structured algebraic specifications*, in: Proc. AFCET Symposium on Mathematics for Computer Science, pp. 93-108, 1982

[Wirsing 86] M. Wirsing: *Structured Algebraic specifications: A Kernel Language*, in: Theoretical Computer Science, *42*, pp. 123-249, 1986

[Wirsing 90] M. Wirsing: *Algebraic Specifications*, in: Handbook of Theoretical Computer Science, (ed.) J. van Leeuwen, North Holland, pp. 676-788, 1990

[Wirth 85] N. Wirth: *Programming in Modula*-2, Springer, Berlin, 1985

Appendix A Basic Specifications

```
aspec BOOL =
  sorts Bool
  cons   true: → Bool,
         false: → Bool,
  opns   not: Bool → Bool,
       .and.: Bool, Bool → Bool,
       .or.: Bool, Bool → Bool,
       .=>.: Bool, Bool → Bool,
       .<=>.: Bool, Bool → Bool,
       eq_Bool: Bool, Bool → Bool
   axioms for all x, y: Bool.
    not(true) = false,
    not(false) = true,
    x and false = false,
    x and true = x,
    false and x = false,
    true and x = x,
    x or false = x,
    x or true = true,
    false or x = x,
    true or x = true,
    x => x = true,
    true => false = false,
    false => x = true,
    x => true = true,
    x <=> x = true,
    true <=> x = x,
    x <=> true = x,
    eq_Bool(true, false) = false,
    eq_Bool(false, true) = false,
    eq_Bool(true, true) = true,
    eq_Bool(false, false) = true
endspec

aspec NAT =
  enrich BOOL by
    sorts Nat
    cons     zero : → Nat,
             .+1 : Nat → Nat,
    opns     .-1 : Nat → Nat,
             .+. : Nat, Nat → Nat,
             .-. : Nat, Nat → Nat,
             .*. : Nat, Nat → Nat,
             eq_Nat : Nat, Nat → Bool,
             ifnat.then.else.fi : Bool, Nat, Nat → Nat,
             .<=. : Nat, Nat → Bool,
             .>=. : Nat, Nat → Bool
    axioms for all x, y: Nat.
      x+1-1 = x,
      x + zero = x,
      x + (y+1) = (x + y)+1,
      zero + x = x,
      (x+1) + y = (x + y)+1,
      zero - x = zero,
      0 -1 = 0,
      (x+1) - zero = x+1,
      (x+1) - (y+1) = x - y,
      x * zero = zero,
      x * (y+1) = x * y + x,
      zero * x = zero,
      (x+1) * y = x * y + x,
      x <= x = true,
      0 <= x = true,
      x+1 <= 0 = false,
      x+1 <= y+1 = x <= y,
      x >= y = y <= x,
      eq_Nat(x, x) = true,
      eq_Nat(0, x+1) =  false,
      eq_Nat(x+1, 0) = false,
      eq_Nat(x+1, y+1) = eq_Nat(x, y),
      ifnat true then x else y fi = x,
      ifnat false then x else y fi = y
endspec

spec ID =
  enrich BOOL by
    sorts Id
    opns eq_Id : Id, Id → Bool
    axioms x, y: Id.
      eq_Id(x, x) = true,
      eq_Id(x, y) = eq_Id(y, x)
endspec

spec ELEM =
  enrich BOOL by
    sorts Elem
    opns    constElem : → Elem,
            eq_Elem : Elem, Elem → Bool,
            ifElem.then.else.fi : Bool, Elem, Elem → Elem
    axioms for all e, e1, e2: Elem.
      eq_Elem(e, e) = true,
      eq_Elem(e1, e2) = eq_Elem(e2, e1),
      ifElem true then e1 else e2 fi = e1,
      ifElem false then e1 else e2 fi = e2
endspec

aspec ENVIRONMENT =
  enrich NAT + ID by
    sorts Env
    cons mtEnv: → Env,
          update: Id, Nat, Env → Env,
          givenEnv: → Env
    opns lookup: Id, Env → Nat
    axioms for all Id, Id1, Id2: Id; n: Nat; e: Env.
    (1)    lookup(Id, update(Id, n, e)) = n,
    (2)    Id1 ≠ Id2 ⇒ lookup(Id1, update(Id2, n, e)) = lookup(Id1, e)[1]
  endspec
```

[1] Note, this conditional equations can also be decribed by an equation using if.then.else.fi .

Appendix B Examples

Appendix B.1 Instantiated Local Dependency Set

The instantiated local dependency set for the term
t ≡ *mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2))
is obtained by instantiating the local dependency set for *CMOBILE* by the term *t*:
{{ *weight*(*mobile*(*mobile*(*occ*(*cube*(1)), *cube*(3)), *cube*(2))) },
{ *weight*(*mobile*(*mobile*(*cube*(1), *occ*(*cube*(3))), *cube*(2))) },
{ *weight*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *occ*(*cube*(2)))) },

{ *weight*(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))),
weight(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
weight(*mobile*(*mobile*(*cube*(1), *cube*(3)), *occ*(*cube*(2)))) },
{ *weight*(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
weight(*mobile*(*mobile*(*occ*(*cube*(1)), *cube*(3)), *cube*(2))),
weight(*mobile*(*mobile*(*cube*(1), *occ*(*cube*(3))), *cube*(2))) },

{ *length*(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))),
leftlength(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))),
rightlength(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))) },
{ *length*(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
leftlength(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
rightlength(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))) },

{ *weight*(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
leftlength(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))),
weight(*mobile*(*mobile*(*cube*(1), *cube*(3)), *occ*(*cube*(2))))
rightlength(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))) },
{ *weight*(*mobile*(*mobile*(*occ*(*cube*(1)), *cube*(3)), *cube*(2))),
leftlength(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
weight(mobile(mobile(cube(1), occ(cube(3))), cube(2)))
rightlength(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))) },

{ *depth*(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))) },

{ *depth*(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))),
depth(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))) },
{ *depth*(*mobile*(*mobile*(*occ*(*cube*(1)), *cube*(3)), *cube*(2))),
depth(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))) },

{ *depth*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *occ*(*cube*(2)))),
depth(*occ*(*mobile*(*mobile*(*cube*(1), *cube*(3)), *cube*(2)))) },
{ *depth*(*mobile*(mobile(*cube*(1), *occ*(*cube*(3))), *cube*(2))),
depth(*mobile*(*occ*(*mobile*(*cube*(1), *cube*(3))), *cube*(2))) },

{ *cmaxdepth*(*mobile*(*mobile*(*occ*(*cube*(1)), *cube*(3)), *cube*(2))),

depth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))) },
{ cmaxdepth(mobile(mobile(cube(1), occ(cube(3))), cube(2))),
depth(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },
{ cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))),
depth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },

{ cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
cmaxdepth(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },
{ cmaxdepth(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
cmaxdepth(mobile(mobile(occ(cube(1)), cube(3)), cube(2))),
cmaxdepth(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },

{ length(mobile(mobile(occ(cube(1)), cube(3)), cube(2))) },
{ length(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },
{ length(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },

{ leftlength(mobile(mobile(occ(cube(1)), cube(3)), cube(2))) },
{ leftlength(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },
{ leftlength(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },

{ rightlength(mobile(mobile(occ(cube(1)), cube(3)), cube(2))) },
{ rightlength(mobile(mobile(cube(1), occ(cube(3))), cube(2))) },
{ rightlength(mobile(mobile(cube(1), cube(3)), occ(cube(2)))) },

{ length(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
depth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))) },
{ length(mobile(occ(mobile(cube(1), cube(3))), cube(2))),
cmaxdepth(occ(mobile(mobile(cube(1), cube(3)), cube(2)))),
depth(mobile(occ(mobile(cube(1), cube(3))), cube(2))) } }

Appendix B.2 Behavioural Implementation Proof

Fact:

It holds:
BEHLMOBILE is implemented by *CMOBILE*2

Proof

For the proof the following complete sets of occurrence terms is used:
CS = { occ(cube(l)), occ(mobile(m1, m2)), sv[mobile(occ(cube(l1)), cube(l2))],
sv[mobile(cube(l1), occ(cube(l2)))], sv[mobile(occ(mobile(m1, m2)), cube(l))],
sv[mobile(mobile(m1, m2), occ(cube(l)))],
sv[mobile(occ(cube(l)), mobile(m1, m2))],
sv[mobile(cube(l), occ(mobile(m1, m2)))],
sv[mobile(occ(mobile(m1, m2)), mobile(m3, m4))],
sv[mobile(mobile(m1, m2), occ(mobile(m3, m4)))] }

The proof that *CS* is a complete set of occurrence terms is left to the reader. The partition for this set of occurrence terms is:

$part(CS) = \{ \{ occ(cube(l)) \}, \{ occ(mobile(m1, m2)) \},$
$\quad \{ \mathrm{sv}[mobile(occ(cube(l1)), cube(l2))], sv[mobile(cube(l1), occ(cube(l2)))] \},$
$\quad \{ \mathrm{sv}[mobile(occ(mobile(m1, m2)), cube(l))],$
$\qquad sv[mobile(mobile(m1, m2), occ(cube(l)))] \},$
$\quad \{ \mathrm{sv}[mobile(occ(cube(l)), mobile(m1, m2))],$
$\qquad sv[mobile(cube(l), occ(mobile(m1, m2)))] \},$
$\quad \{ \mathrm{sv}[mobile(occ(mobile(m1, m2)), mobile(m3, m4))],$
$\qquad sv[mobile(mobile(m1, m2), occ(mobile(m3, m4)))] \} \}$

The observable attributes, i.e. the attributes with observable result sort are

$Attr = \{ leftlength, rightlength, length, depth, cmaxdepth \}$

I.e. it must hold:

(1) $Solutions^{CMOBILE2}(\{occ(cube(l))\}) \supseteq Solutions^{BEHLMOBILE}(\{occ(cube(l))\})$

(2) $Solutions^{CMOBILE2}(\{occ(mobile(m1, m2))\}) \supseteq$
$\quad Solutions^{BEHLMOBILE}(\{occ(mobile(m1, m2))\})$

(3) $Solutions^{CMOBILE2}(\{sv[mobile(occ(cube(l1)), cube(l2))],$
$\qquad sv[mobile(cube(l1), occ(cube(l2)))]\}) \supseteq$
$\quad Solutions^{BEHLMOBILE}(\{sv[mobile(occ(cube(l1)), cube(l2))],$
$\qquad sv[mobile(cube(l1), occ(cube(l2)))]\})$

(4) $Solutions^{CMOBILE2}(\{sv[mobile(occ(cube(l)), mobile(m1, m2))],$
$\qquad sv[mobile(cube(l), occ(mobile(m1, m2)))]\}) \supseteq$
$\quad Solutions^{BEHLMOBILE}(\{sv[mobile(occ(cube(l)), mobile(m1, m2))],$
$\qquad sv[mobile(cube(l), occ(mobile(m1, m2)))]\})$

(5) $Solutions^{CMOBILE2}(\{sv[mobile(occ(mobile(m1, m2)), cube(l))],$
$\qquad sv[mobile(mobile(m1, m2), occ(cube(l)))]\}) \supseteq$
$\quad Solutions^{BEHLMOBILE}(\{sv[mobile(occ(mobile(m1, m2)), cube(l))],$
$\qquad sv[mobile(mobile(m1, m2), occ(cube(l)))]\})$

(6) $Solutions^{CMOBILE2}(\{sv[mobile(occ(mobile(m1, m2)), mobile(m3, m4))],$
$\qquad sv[mobile(mobile(m1, m2), occ(mobile(m3, m4)))]\}) \supseteq$
$\quad Solutions^{BEHLMOBILE}(\{sv[mobile(occ(mobile(m1, m2)), mobile(m3, m4))],$
$\qquad sv[mobile(mobile(m1, m2), occ(mobile(m3, m4)))]\})$

For the proof of the correctness of the implementation relation the following fact (*) is necessary:

$\forall sv[z_{Mobile}]: Mobile \rightarrow Mobile\ \forall x: Mobile.$
$\qquad weight^{BEHLMOBILE}(sv[occ(x)]) = weight^{CMOBILE2}(sv[occ(x)]) * 2$

which can be shown using a complete set of occurrence terms of sort *Mobile*. The proof is left to the reader.

Note, that the proofs are extremely detailed and a lot of work can be done by a system.

Proof

Proof obligation (1):

$Solutions^{CMOBILE2}(\{ occ(cube(l)) \}) \supseteq Solutions^{BEHLMOBILE}(\{ occ(cube(l)) \})$

proof:

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\ \mathit{occ}(\mathit{cube}(l))\ \}) =$

$\{\ \sigma \mid \mathit{Mod}(\mathit{CMOBILE2}) \models_{\text{attr}} \mathit{leftlength}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_1),$

$\quad \mathit{rightlength}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_2),\ \mathit{length}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_3),$

$\quad \mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_4),\ \mathit{cmaxdepth}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_5)\ \}$

$= \{\ [\ x_1 / 0,\ x_2 / 0,\ x_3 / 0,\ x_4 / 1,\ x_5 / 1\ ]\ \}$

$\supseteq \{\ [\ x_1 / 0,\ x_2 / 0,\ x_3 / 0,\ x_4 / 1,\ x_5 / 1\ ]\ \}$

$= \{\ \sigma \mid \mathit{Mod}(\mathit{BEHLMOBILE}) \models_{\text{attr}} \mathit{leftlength}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_1),$

$\quad \mathit{rightlength}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_2),\ \mathit{length}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_3),$

$\quad \mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_4),\ \mathit{cmaxdepth}(\mathit{occ}(\mathit{cube}(l))) = \sigma(x_5)\ \}$

$= \mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\ \mathit{occ}(\mathit{cube}(l))\ \})$

Since in *CMOBILE*2 holds

$\mathit{leftlength}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{rightlength}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{length}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = 1,$
$\mathit{cmaxdepth}(\mathit{occ}(\mathit{cube}(l))) = \mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = 1$

and in *BEHLMOBILE* holds:

$\mathit{leftlength}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{rightlength}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{length}(\mathit{occ}(\mathit{cube}(l))) = 0,$
$\mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = 1,$
$\mathit{cmaxdepth}(\mathit{occ}(\mathit{cube}(l))) = \mathit{depth}(\mathit{occ}(\mathit{cube}(l))) = 1$

Proof obligation (2):

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\ \mathit{occ}(\mathit{mobile}(m1,\ m2))\ \}) \supseteq$
$\quad \mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\ \mathit{occ}(\mathit{mobile}(m1,\ m2))\ \})$

induction assertion:

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\ \mathit{mobile}(\mathit{occ}(m1),\ m2),\ \mathit{mobile}(m1,\ \mathit{occ}(m2))\ \}) \subseteq$
$\quad \mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\ \mathit{mobile}(\mathit{occ}(m1),\ m2),\ \mathit{mobile}(m1,\ \mathit{occ}(m2))\ \})$

proof:

$\mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\mathit{occ}(\mathit{mobile}(m1,\ m2))\}) =$

$\{\ \sigma \mid \mathit{Mod}(\mathit{BEHLMOBILE}) \models_{\text{attr}} \mathit{leftlength}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) = \sigma(x_1),$

$\quad \mathit{rightlength}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) = \sigma(x_2),$

$\quad \mathit{length}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) = \sigma(x_3),$

$\quad \mathit{depth}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) = \sigma(x_4),$

$\quad \mathit{cmaxdepth}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) = \sigma(x_5)\ \}$

$= \{\ [\ x_1 / \mathit{length}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) * \mathit{weight}^{\mathit{BEHLMOBILE}}(\mathit{mobile}(m1,\ \mathit{occ}(m2))) /$

$\qquad \mathit{weight}^{\mathit{BEHLMOBILE}}(\mathit{occ}(\mathit{mobile}(m1,\ m2))),$

$\quad x_2 / \mathit{length}(\mathit{occ}(\mathit{mobile}(m1,\ m2))) * \mathit{weight}^{\mathit{BEHLMOBILE}}(\mathit{mobile}(\mathit{occ}(m1),\ m2)) /$

$\qquad \mathit{weight}^{\mathit{BEHLMOBILE}}(\mathit{occ}(\mathit{mobile}(m1,\ m2))),\ x_4 / 1,$

$\quad x_5 / \mathit{max}(\mathit{cmaxdepth}(\mathit{mobile}(\mathit{occ}(m1),\ m2)),\ \mathit{cmaxdepth}(\mathit{mobile}(m1,\ \mathit{occ}(m2))))\ \}$

Putting this solutions into the attribute equations for the considered term in *CMOBILE*2 and showing that the attribute equations with these solutions are valid. The inclusion follows immediately.

Proof obligation (3):

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))],$
$\quad \mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]\}) \supseteq$
$\mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\mathrm{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))],$
$\quad \mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]\})$

induction assertion:

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]\})$
$\subseteq \mathit{Solutions}^{\mathit{BEHLMOBILE}}(\{\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2))]\})$

proof:

$\mathit{Solutions}^{\mathit{CMOBILE2}}(\{\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))],$
$\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]\}) =$
$\{ \sigma \mid \mathit{Mod}(\mathit{CMOBILE2}) \models_{\mathrm{attr}}$
$\quad \mathit{leftlength}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_1),$
$\quad \mathit{rightlength}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_2),$
$\quad \mathit{length}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_3),$
$\quad \mathit{depth}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_4),$
$\quad \mathit{cmaxdepth}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_5),$
$\quad \mathit{leftlength}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_6),$
$\quad \mathit{rightlength}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) =\sigma(x_7),$
$\quad \mathit{length}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_8),$
$\quad \mathit{depth}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_9),$
$\quad \mathit{cmaxdepth}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_{10}) \}$
$= \{ [x_1 / 0, x_2 / 0, x_3 / 0, x_4 / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 ,$
$\quad x_5 / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 ,$
$\quad x_6 / 0, x_7 / 0, x_8 / 0, x_9 / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 ,$
$\quad x_{10} / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 \}$
$\supseteq \{ [x_1 / 0, x_2 / 0, x_3 / 0, x_4 / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)) + 1 ,$
$\quad x_5 / \mathit{depth}(\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))), x_6 / 0, x_7 / 0, x_8 / 0,$
$\quad x_9 / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 ,$
$\quad x_{10} / \mathit{depth}(\mathit{sv}[\mathit{occ}(\mathit{mobile}(\mathit{cube}(l1), \mathit{cube}(l2)))]) + 1 \}$
$= \{ \sigma \mid \mathit{Mod}(\mathit{BEHLMOBILE}) \models_{\mathrm{attr}}$
$\quad \mathit{leftlength}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_1),$
$\quad \mathit{rightlength}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_2),$
$\quad \mathit{length}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_3),$
$\quad \mathit{depth}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_4),$
$\quad \mathit{cmaxdepth}(\mathit{sv}[\mathit{mobile}(\mathit{occ}(\mathit{cube}(l1)), \mathit{cube}(l2))]) = \sigma(x_5),$
$\quad \mathit{leftlength}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_6),$
$\quad \mathit{rightlength}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) =\sigma(x_7),$
$\quad \mathit{length}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_8),$
$\quad \mathit{depth}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_9),$
$\quad \mathit{cmaxdepth}(\mathit{sv}[\mathit{mobile}(\mathit{cube}(l1), \mathit{occ}(\mathit{cube}(l2)))]) = \sigma(x_{10}) \}$

= *Solutions*^{BEHLMOBILE}(\{*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))],
sv[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]\})
Since in *CMOBILE*2 holds
leftlength(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
rightlength(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
length(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
depth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
cmaxdepth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
leftlength(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
rightlength(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
length(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
depth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
cmaxdepth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1
and in *BEHLMOBILE* holds:
leftlength(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
rightlength(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
length(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) = 0,
depth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
cmaxdepth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*mobile*(*occ*(*cube*(*l*1)), *cube*(*l*2))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
leftlength(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
rightlength(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
length(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) = 0,
depth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1,
cmaxdepth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*mobile*(*cube*(*l*1), *occ*(*cube*(*l*2)))]) =
depth(*sv*[*occ*(*mobile*(*cube*(*l*1), *cube*(*l*2)))]) + 1

Proof obligation (4):

Solutions^{CMOBILE2}(\{*sv*[*mobile*(*occ*(*cube*(*l*)), *mobile*(*m*1, *m*2))],
sv[*mobile*(*cube*(*l*), *occ*(*mobile*(*m*1, *m*2)))]\}) ⊇
Solutions$^{BEHI\ MOBILE}$(\{sv[*mobile*(*occ*(*cube*(*l*)), *mobile*(*m*1, *m*2))],
sv[*mobile*(*cube*(*l*), *occ*(*mobile*(*m*1, *m*2)))]\})

induction assertion:

Solutions^{CMOBILE2}(\{*sv*[*occ*(*mobile*(*cube*(*l*), *mobile*(*m*1, *m*2)))],
sv[*mobile*(*cube*(*l*), *mobile*(*occ*(*m*1), *m*2))],
sv[*mobile*(*cube*(*l*), *mobile*(*m*1, *occ*(*m*2)))]\}) ⊇
Solutions^{BEHLMOBILE}(\{*sv*[*occ*(*mobile*(*cube*(*l*), *mobile*(*m*1, *m*2)))],

sv[*mobile*(*cube*(*l*), *mobile*(*occ*(*m*1), *m*2))],
sv[*mobile*(*cube*(*l*), *mobile*(*m*1, *occ*(*m*2)))]})

proof:

similar to (3)

Proof obligation (5):
$\mathit{Solutions}^{\mathit{CMOBII\,E2}}$({*sv*[*mobile*(*occ*(*mobile*(*m*1, *m*2)), *cube*(*l*))],
sv[*mobile*(*mobile*(*m*1, *m*2), *occ*(*cube*(*l*)))]}) $\supseteq$
$\mathit{Solutions}^{\mathit{BEHLMOBILE}}$({*sv*[*mobile*(*occ*(*mobile*(*m*1, *m*2)), *cube*(*l*))],
sv[*mobile*(*mobile*(*m*1, *m*2), *occ*(*cube*(*l*)))]})

proof
analogous to proof obligation (4)

Proof obligation (6):
analogous to proof obligation (4) and (5) ♦

Appendix B.3 Solving Existentially Quantified Formulae

Fact to be solved:
$\exists\, sv_{Call,\ Task \rightarrow Task} \cdot \exists\, cm_{ConferenceMenu} \cdot$
stateafter(*mkDialog*(*sv*[*occ*(*mkCall*(*CALL*, *mkEnterTNumber*(*nr*))), *cm*])) =
mkState(*mkCon*(*nr*, *telephoning*), *mtCon*)
and
stateafter(*mkDialog*(*sv*[*mkCall*(*CALL*, *mkEnterTNumber*(*nr*)), *occ*(*cm*)])) =
mkState(*mkCon*(*nr*, *telephoning*), *mkCon*(*nr*2, *telephoning*))
A possible derivation resulting in the least solution, i.e. in such a solution that the depth of *sv* is minimal can be obtained as follows.
The start configuration is
({ *x*1 = *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*),
*x*2 = *mkState*(*mkCon*(*nr*, *telephoning*), *mkCon*(*nr*2, *telephoning*)) },
[],
{ *x*1 = *stateafter*(*mkDialog*(*sv*[*occ*(*mkCall*(*CALL*, *mkEnterTNumber*(*nr*))), *cm*])),
*x*2 = *stateafter*(*mkDialog*(*sv*[*mkCall*(*CALL*, *mkEnterTNumber*(*nr*)), *occ*(*cm*)])})
$\vdash_{(\mathrm{N3,\ ISDN\text{-}Attribution\ (3)})}$[2]
({ *x*1 = *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*),
*x*2 = *mkState*(*mkCon*(*nr*, *telephoning*), *mkCon*(*nr*2, *telephoning*)),
*x*2 = *conference*(*x*3) },
[*cm* / *CONFERENCE*],
{ *x*1 = *stateafter*(*mkDialog*(*sv*[*occ*(*mkCall*(*CALL*, *mkEnterTNumber*(*nr*))),
CONFERENCE])),
*x*2 = *stateafter*(*mkDialog*(*sv*[*mkCall*(*CALL*, *mkEnterTNumber*(*nr*)),
occ(*CONFERENCE*)])),
*x*3 = *statebefore*(*mkDialog*(*sv*[*mkCall*(*CALL*, *mkEnterTNumber*(*nr*)),
occ(*CONFERENCE*)])) })

[2] By *ISDN-Attribution* (*i*) the i-th axiom of the specification *ISDN-Attribution* is denoted.

$$\vdash_{\text{(N3, ISDN-Attribution (18))}}$$

$$\begin{aligned}
(\{\ & x1 = mkState(mkCon(nr, telephoning), mtCon),\\
& x2 = mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning)),\\
& x2 = conference(x3),\\
& x3 = x5\ \},\\
[\ & cm\ /\ CONFERENCE,\\
& sv\ /\ sv1[mkConference(z_{Call}, x4, z_{ConferenceMenu})]\ ],\\
\{\ & x1 = stateafter(mkDialog(sv1[mkConference(occ(mkCall(CALL,\\
& \quad mkEnterTNumber(nr))), x4, CONFERENCE])),\\
& x2 = stateafter(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), x4, occ(CONFERENCE))])),\\
& x3 = statebefore(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), x4, occ(CONFERENCE))])),\\
& x5 = stateafter(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), occ(x4), CONFERENCE)]))\ \}\ )
\end{aligned}$$

$$\vdash_{\text{(N3, ISDN-Attribution (15))}}$$

$$\begin{aligned}
(\{\ & x1 = mkState(mkCon(nr, telephoning), mtCon),\\
& x2 = mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning)),\\
& x2 = conference(x3),\\
& x3 = x5,\\
& x5 = secondCall(x6, x7)\ \},\\
[\ & cm\ /\ CONFERENCE,\\
& sv\ /\ sv1[mkConference(z_{Call}, mkSecondCall(SECONDCALL,\\
& \qquad mkEnterTNumber(x8)), z_{ConferenceMenu})],\\
& x4\ /\ mkSecondCall(SECONDCALL, mkEnterTNumber(x8))\ ],\\
\{\ & x1 = stateafter(mkDialog(sv1[mkConference(occ(mkCall(CALL,\\
& \quad mkEnterTNumber(nr))), mkSecondCall(SECONDCALL,\\
& \quad mkEnterTNumber(x8)), CONFERENCE)])),\\
& x2 = stateafter(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)),\\
& \quad mkSecondCall(SECONDCALL, mkEnterTNumber(x8)),\\
& \quad occ(CONFERENCE))])),\\
& x3 = statebefore(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), mkSecondCall(SECONDCALL,\\
& \quad mkEnterTNumber(x8)), occ(CONFERENCE))])),\\
& x5 = stateafter(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), occ(mkSecondCall(SECONDCALL,\\
& \quad mkEnterTNumber(x8))), CONFERENCE)])),\\
& x7 = statebefore(mkDialog(sv1[mkConference(mkCall(CALL,\\
& \quad mkEnterTNumber(nr)), occ(mkSecondCall(SECONDCALL,\\
& \quad mkEnterTNumber(x8))), CONFERENCE)]))\ \}\ )
\end{aligned}$$

$$\vdash_{\text{(N3, ISDN-Attribution (17))}}$$

$$\begin{aligned}
(\{\ & x1 = mkState(mkCon(nr, telephoning), mtCon),\\
& x2 = mkState(mkCon(nr, telephoning), mkCon(nr2, telephoning)),\\
& x2 = conference(x3),\\
& x3 = x5,\\
& x5 = secondCall(x6, x1)\ \},
\end{aligned}$$

[*cm* / *CONFERENCE*,
sv / *sv*1[*mkConference*(z_{Call}, *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), $z_{ConferenceMenu}$)],
*x*4 / *mkSecondCall*(*SECONDCALL*, *mkEnterTNumber*(*x*8)),
*x*7 / *x*1],
{ *x*1 = *stateafter*(*mkDialog*(*sv*1[*mkConference*(*occ*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*))), *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), *CONFERENCE*)])),
*x*2 = *stateafter*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)),
mkSecondCall(*SECONDCALL*, *mkEnterTNumber*(*x*7)),
occ(*CONFERENCE*))])),
*x*3 = *statebefore*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)), *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), *occ*(*CONFERENCE*))])),
*x*5 = *stateafter*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)), *occ*(*mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8))), *CONFERENCE*)])),
*x*1 = *statebefore*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)), *occ*(*mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8))), *CONFERENCE*])) })

$\vdash_{(N2)}$

({ *x*2 = *mkState*(*mkCon*(*nr*, *telephoning*), *mkCon*(*nr*2, *telephoning*)),
*x*2 = *conference*(*x*3),
*x*3 = *x*5,
*x*5 = *secondCall*(*x*6, *x*1) },
[*cm* / *CONFERENCE*,
sv / *sv*1[*mkConference*(z_{Call}, *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), $z_{ConferenceMenu}$)],
*x*4 / *mkSecondCall*(*SECONDCALL*, *mkEnterTNumber*(*x*8)),
*x*7 / *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*),
*x*1 / *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*)],
{ *mkState*(*mkCon*(*nr*, *telephoning*), *mtCon*) =
stateafter(*mkDialog*(*sv*1[*mkConference*(*occ*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*))), *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), *CONFERENCE*)])),
*x*2 = *stateafter*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)),
mkSecondCall(*SECONDCALL*, *mkEnterTNumber*(*x*8)),
occ(*CONFERENCE*))])),
*x*3 = *statebefore*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)), *mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8)), *occ*(*CONFERENCE*)])),
*x*5 = *stateafter*(*mkDialog*(*sv*1[*mkConference*(*mkCall*(*CALL*,
mkEnterTNumber(*nr*)), *occ*(*mkSecondCall*(*SECONDCALL*,
mkEnterTNumber(*x*8))), *CONFERENCE*)])),
mkState(*mkCon*(*nr*, *telephoning*), *mtCon*) =

$$\begin{aligned}&\mathit{statebefore}(\mathit{mkDialog}(\mathit{sv}1[\mathit{mkConference}(\mathit{mkCall}(\mathit{CALL},\\ &\mathit{mkEnterTNumber}(\mathit{nr})),\ \mathit{occ}(\mathit{mkSecondCall}(\mathit{SECONDCALL},\\ &\mathit{mkEnterTNumber}(x8))),\ \mathit{CONFERENCE})]))\ \}\ )\end{aligned}$$

As next steps the attribute equations have only be solved without complicated transformations.

Appendix B.4 Proving the Correctness of the Compiler Specifications

Fact:

It holds:

$$\begin{aligned}&\forall\ \mathit{sv}_{\mathit{Instr} \rightarrow \mathit{Instr}}.\ \forall\ \mathit{sv}'_{\mathit{Expr} \rightarrow \mathit{Expr}}.\ \forall\ e_{\mathit{Expr}}.\\ &\quad \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(e)]))]) =\\ &\qquad \mathit{push}(\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(e)]))]),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(e)]))]))\end{aligned}$$

Proof

This property is shown using the following complete sets of occurrence terms:

$CS_{sv} = \{\ \mathit{sv}_{\mathit{Instr} \rightarrow \mathit{Instr}}\ \}$

$CS_{sv'} = \{\ \mathit{sv}'_{\mathit{Expr} \rightarrow \mathit{Expr}}\ \}$

$CS_e = \{\ \mathit{natexpr}(n), \mathit{idexpr}(\mathit{id}), \mathit{comp}(e1, \mathit{add}, e2), \mathit{comp}(e1, \mathit{sub}, e2),\ \mathit{comp}(e1, \mathit{mult}, e2)\ \}$

***natexpr*(*n*):**

$$\begin{aligned}&\mathit{lhs}\text{: } \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{natexpr}(n))]))]) = \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))]) =\\ &\quad \mathit{push}(n,\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))]))\end{aligned}$$

$$\begin{aligned}&\mathit{rhs}\text{: } \mathit{push}(\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{natexpr}(n))]))]),\\ &\qquad \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{natexpr}(n))]))])) =\\ &\quad \mathit{push}(\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))]),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))])) =\\ &\quad \mathit{push}(\mathit{top}(\mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))])),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))])) =\\ &\quad \mathit{push}(n,\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{NST}(n))]))\end{aligned}$$

***idexpr*(*id*):**

$$\begin{aligned}&\mathit{lhs}\text{: } \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]))]) = \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))]) =\\ &\quad \mathit{push}(\mathit{lookup}(\mathit{id},\ \mathit{givenEnv}),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))]))\end{aligned}$$

$$\begin{aligned}&\mathit{rhs}\text{: } \mathit{push}(\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]))]),\\ &\qquad \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]))])) =\\ &\quad \mathit{push}(\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))]),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))])) =\\ &\quad \mathit{push}(\mathit{top}(\mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))])),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))])) =\\ &\quad \mathit{push}(\mathit{lookup}(\mathit{id},\ \mathit{givenEnv}),\ \mathit{stackb}(\mathit{sv}[\mathit{occ}(\mathit{IST}(\mathit{id}))]))\end{aligned}$$

***comp*(*e*1, *add*, *e*2):**

$$\begin{aligned}&\mathit{lhs}\text{: } \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{occ}(\mathit{comp}(e1,\ \mathit{add},\ e2))]))]) =\\ &\quad \mathit{stacka}(\mathit{sv}[\mathit{occ}(\mathit{code}(\mathit{sv}'[\mathit{comp}(\mathit{occ}(e1),\ \mathit{add},\ e2)])\ ;\\ &\qquad \mathit{code}(\mathit{sv}'[\mathit{comp}(e1,\ \mathit{add},\ \mathit{occ}(e2))]))])\ ;\mathit{ADD})]) =\\ &\quad \mathit{stacka}(\mathit{sv}[\mathit{code}(\mathit{sv}'[\mathit{comp}(\mathit{occ}(e1),\ \mathit{add},\ e2)]);\end{aligned}$$

code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]) =
push(top(pop(stackb(sv[code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]))) +
top(stackb(sv[code(sv'[comp(occ(e1), add, e2)]);
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]),
pop(pop(stackb(sv[code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]))) =
push(...,
pop(pop(stacka(sv[code(sv'[comp(occ(e1), add, e2)]) ;
occ(code(sv'[comp(e1, add, occ(e2))]))])) ; ADD]))) =
push(...,
pop(pop(stacka(sv[code(sv'[comp(occ(e1), add, e2)]) ;
occ(code(sv'[comp(e1, add, occ(e2))]))])) ; ADD]))) = (induction assertion)
push(...,
pop(pop(push(..., stackb(sv[code(sv'[comp(occ(e1), add, e2)]);
occ(code(sv'[comp(e1, add, occ(e2))]))])) ; ADD])))) =
push(...,
pop(stackb(sv[code(sv'[comp(occ(e1), add, e2)]) ;
occ(code(sv'[comp(e1, add, occ(e2))]))])) ; ADD])) =
push(...,
pop(stacka(sv[occ(code(sv'[comp(occ(e1), add, e2)])) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD])) = (induction assertion)
push(...,
pop(push(..., stackb(sv[occ(code(sv'[comp(occ(e1), add, e2)])) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD]))) =
push(top(pop(stackb(sv[code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]))) +
top(stackb(sv[code(sv'[comp(occ(e1), add, e2)]);
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]),
stackb(sv[occ(code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD)]))

rhs: *push(value(sv[occ(code(sv'[occ(comp(e1, add, e2))]))]),*
stackb(sv[occ(code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD)])) =
push(top(stacka(sv[occ(code(sv'[occ(comp(e1, add, e2))]))]))])),
stackb(sv[occ(code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD)])) = ... as *lhs* ... =
push(top(pop(stackb(sv[code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]))) +
top(stackb(sv[code(sv'[comp(occ(e1), add, e2)]);
code(sv'[comp(e1, add, occ(e2))]))]) ; occ(ADD)]),
stackb(sv[occ(code(sv'[comp(occ(e1), add, e2)]) ;
code(sv'[comp(e1, add, occ(e2))]))]) ; ADD)]))

***comp*(*e*1, *sub*, *e*2), *comp*(*e*1, *mult*, *e*2):**
analogous to *comp*(*e*1, *add*, *e*2). ◆

Fact:

It holds:

$$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.\ value(sv[occ(e)]) = value(occ(code(sv[occ(e)])))$$

Proof

This property is shown using the following complete set of occurrence terms:

$$\{\ sv[occ(natexpr(n))],\ sv[occ(idexpr(id))],\ sv[occ(comp(e1,\ add,\ e2))],$$
$$sv[occ(comp(e1,\ sub,\ e2))],\ sv[occ(comp(e1,\ mult,\ e2))]\ \}$$

$sv[occ(natexpr(n))]$:

lhs: $value(sv[occ(natexpr(n))]) = n$

rhs: $value(occ(code(sv[occ(natexpr(n))]))) = value(occ(NST(n))) =$
$top(stacka(occ(NST(n)))) = top(push(n,\ stackb(occ(NST(n))))) = n$

$sv[occ(idexpr(id))]$:

lhs: $value(sv[occ(idexpr(id))]) = lookup(n,\ env(occ(sv[idexpr(id)]))) =$
$lookup(id,\ givenEnv)$

rhs: $value(occ(code(sv[occ(idexpr(id))]))) = value(occ(IST(id))) =$
$top(stacka(occ(IST(id)))) =$
$top(push(lookup(id,\ env(occ(sv[idexpr(id)]))),\ stackb(occ(IST(id))))) =$
$lookup(id,\ env(occ(sv[idexpr(id)]))) = lookup(id,\ givenEnv)$

$sv[occ(comp(e1,\ add,\ e2))]$:

lhs: $value(sv[occ(comp(e1,\ add,\ e2))]) =$
$value(sv[comp(occ(e1),\ add,\ e2)]) + value(sv[comp(e1,\ add,\ occ(e2))])$

rhs: $value(occ(code(sv[occ(comp(e1,\ add,\ e2))]))) =$
$value(occ(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad code(sv[comp(e1,\ add,\ occ(e2))])\ ; ADD)) =$
$top(stacka(occ(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad code(sv[comp(e1,\ add,\ occ(e2))])\ ; ADD))) =$
$top(stacka(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad code(sv[comp(e1,\ add,\ occ(e2))])\ ;\ occ(ADD))) =$
$top(pop(stackb(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad code(sv[comp(e1,\ add,\ occ(e2))])\ ;\ occ(ADD)))) +$
$\quad top(stackb(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad code(sv[comp(e1,\ add,\ occ(e2))])\ ;\ occ(ADD))) =$
$top(pop(stacka(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad occ(code(sv[comp(e1,\ add,\ occ(e2))]))\ ; ADD))) +$
$\quad top(stacka(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad occ(code(sv[comp(e1,\ add,\ occ(e2))]))\ ; ADD)) =$
$top(pop(push(value(sv[comp(occ(e1),\ add,\ e2)]),$
$\quad stackb(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad occ(code(sv[comp(e1,\ add,\ occ(e2))]))\ ; ADD))))) +$
$\quad top(push(value(sv[comp(occ(e1),\ add,\ e2)]),$
$\quad stackb(code(sv[comp(occ(e1),\ add,\ e2)])\ ;$
$\quad occ(code(sv[comp(e1,\ add,\ occ(e2))]))\ ; ADD))) =$
$top(stacka(occ(code(sv[comp(occ(e1),\ add,\ e2)]))\ ;$

code(sv[comp(e1, add, occ(e2))]) ; ADD))) +
value(sv[comp(occ(e1), add, e2)]) =
top(push(value(sv[comp(e1, add, occ(e2)]),
stackb(occ(code(sv[comp(occ(e1), add, e2)]) ;
code(sv[comp(e1, add, occ(e2))]) ; ADD))) +
value(sv[comp(occ(e1), add, e2)]) =
value(sv[comp(occ(e1), add, e2)]) + value(sv[comp(e1, add, occ(e2)])

sv[occ(comp(e1, sub, e2))], sv[occ(comp(e1, mult, e2))]:

analogous to *add.*

Fact:

The following equation holds in the specifications *NOPT* and *OPT*

$\forall\ sv_{Expr \rightarrow Expr}.\ \forall\ e_{Expr}.$
isdef(usedregb(sv[occ(e)])) = true ♦

Proof

The proof is done using the following complete set of occurrence terms:

{ *sv[occ(natexpr(n))], sv[occ(idexpr(id))], sv[occ(comp(e1, add, e2))],*
sv[occ(comp(e1, sub, e2))], sv[occ(comp(e1, mult, e2))] }

sv[occ(natexpr(n))]:
In the specification *OPT*:
isdef(usedregb^{OPT}*(sv[occ(natexpr(n))])) =*
isdefined(usedregb^{OPT}*(sv[occ(natexpr(n))]),*
regIs(usedregb^{OPT}*(sv[occ(natexpr(n))]), n)) = true*

In the specification *NOPT*:
identical derivation

sv[occ(idexpr(id))]:
In the specification *OPT*:
isdef(usedregb^{OPT}*(sv[occ(idexpr(id))])) =*
isdefined(usedregb^{OPT}*(sv[occ(idexpr(id))]),*
regIs(usedregb^{OPT}*(sv[occ(idexpr(id))]), lookup(id, givenEnv))) = true*

In the specification *NOPT*:
identical derivation

sv[occ(comp(e1, add, e2))]:
In the specification *OPT*:

case analysis over
less(needed(sv[comp(occ(e1), add, e2)]), needed(sv[comp(e1, add, occ(e2))])) = false:
isdef(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))])) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[occ(comp(e1, add, e2))])) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*

seq(code^{OPT}*(sv[comp(occ(e1), add, e2)]),*
seq(code^{OPT}*(sv[comp(e1, add, occ(e2))]),*
ADD(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
usedregb^{OPT}*(sv[comp(occ(e1), add, e2)]),*
usedregb^{OPT}*(sv[comp(e1, add, occ(e2))]))))) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(occ(e1), add, e2)]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(e1, add, occ(e2))]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
ADD(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
usedregb^{OPT}*(sv[comp(occ(e1), add, e2)]),*
usedregb^{OPT}*(sv[comp(e1, add, occ(e2))])) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(occ(e1), add, e2)]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(e1, add, occ(e2))]))*
or (isdef(usedregb^{OPT}*(sv[comp(occ(e1), add, e2)])) and*
isdef(usedregb^{OPT}*(sv[comp(e1, add, occ(e2))]))) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(occ(e1), add, e2)]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(e1, add, occ(e2))]))*
or (true and true) = true

less(needed(sv[comp(occ(e1), add, e2)]), needed(sv[comp(e1, add, occ(e2))])) = true:
lhs:
isdef(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))])) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[occ(comp(e1, add, e2))])) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
seq(code^{OPT}*(sv[comp(e1, add, occ(e2))]),*
seq(code^{OPT}*(sv[comp(occ(e1), add, e2)]),*
ADD(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
usedregb^{OPT}*(sv[comp(occ(e1), add, e2)]),*
usedregb^{OPT}*(sv[comp(e1, add, occ(e2))]))))) =*
isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(e1, add, occ(e2))]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
code^{OPT}*(sv[comp(occ(e1), add, e2)]))*
or isdefined(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
ADD(usedregb^{OPT}*(sv[occ(comp(e1, add, e2))]),*
usedregb^{OPT}*(sv[comp(occ(e1), add, e2)]),*
usedregb^{OPT}*(sv[comp(e1, add, occ(e2))]))) =*

$$\begin{array}{l}
\mathit{isdefined}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\quad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\quad \mathit{or}\ \mathit{isdefined}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) \\
\quad \mathit{or}\ (\mathit{isdef}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]))\ \mathit{and} \\
\qquad\qquad \mathit{isdef}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])))) = \\
\mathit{isdefined}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\quad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\quad \mathit{or}\ \mathit{isdefined}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) \\
\quad \mathit{or}\ (\mathit{true}\ \mathit{and}\ \mathit{true}) = \mathit{true}
\end{array}$$

In the specification *NOPT*:
identical derivation as for the case
$\mathit{less}(\mathit{needed}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]), \mathit{needed}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) = \mathit{false}$

***sv*[*occ*(*comp*(*e*1, *sub*, *e*2))] and *sv*[*occ*(*comp*(*e*1, *mult*, *e*2))]:**

analogous to *add* ◆

Fact:

The following equation holds between the specifications *SOURCE* and *NOPT*

$$\begin{array}{l}
\forall\ \mathit{sv}_{Expr \rightarrow Expr}.\ \forall\ e_{Expr}. \\
\mathit{value}(\mathit{sv}[\mathit{occ}(e)]) = \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(e)]))
\end{array}$$ ◆

Proof

The proof is done using the following complete set of occurrence terms:

$$\begin{array}{l}
\{\ \mathit{sv}[\mathit{occ}(\mathit{natexpr}(n))], \mathit{sv}[\mathit{occ}(\mathit{idexpr}(\mathit{id}))], \mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))], \\
\quad \mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{sub}, e2))], \mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{mult}, e2))]\ \}
\end{array}$$

***sv*[*occ*(*natexpr*(*n*))]:**

$$\begin{array}{l}
\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{natexpr}(n))]) = n = \\
\quad \mathit{regval}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{natexpr}(n))]), \\
\qquad \mathit{regIs}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{natexpr}(n))]), n)) = \\
\quad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{natexpr}(n))]))
\end{array}$$

***sv*[*occ*(*idexpr*(*id*))]:**

$$\begin{array}{l}
\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]) = \\
\quad \mathit{lookup}(\mathit{id}, \mathit{givenEnv}) = \\
\quad \mathit{regval}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]), \\
\qquad \mathit{regIs}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]), \mathit{lookup}(\mathit{id}, \mathit{givenEnv}))) = \\
\quad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{idexpr}(\mathit{id}))]))
\end{array}$$

***sv*[*occ*(*comp*(*e*1, *add*, *e*2))]:**

lhs:

$$\begin{array}{l}
\mathit{value}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]) = \\
\quad \mathit{value}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]) + \mathit{value}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])
\end{array}$$

$val(usedregb^{NOPT}(sv[comp(occ(e1), add, e2)])) +$
$\quad val(usedregb^{NOPT}(sv[comp(e1, add, occ(e2))]))$

rhs:

$val(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))])) =$
$\quad regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad code^{NOPT}(sv[occ(comp(e1, add, e2))])) =$
$\quad regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad seq(code^{NOPT}(sv[comp(occ(e1), add, e2)]),$
$\qquad\quad seq(code^{NOPT}(sv[comp(e1, add, occ(e2))]),$
$\qquad\qquad ADD(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\qquad\quad usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\qquad\quad usedregb^{NOPT}(sv[comp(e1, add, occ(e2))]))))) =$
$\quad regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad code^{NOPT}(sv[comp(occ(e1), add, e2)]))$
$\qquad \& \; regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\quad code^{NOPT}(sv[comp(e1, add, occ(e2))]))$
$\qquad \& \; regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\quad ADD(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\qquad usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),$
$\qquad\qquad usedregb^{NOPT}(sv[comp(e1, add, occ(e2))]))$
$\quad val(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))])) +$
$\qquad val(usedregb^{NOPT}(sv[comp(e1, add, occ(e2))])) =$
$\quad val(usedregb^{NOPT}(sv[comp(occ(e1), add, e2)])) +$
$\qquad val(\mathrm{usedregb}^{\mathrm{NOPT}}(sv[comp(e1, add, occ(e2))]))$

***sv*[*occ*(*comp*(*e*1, *sub*, *e*2))] and *sv*[*occ*(*comp*(*e*1, *mult*, *e*2))]:**

analogous to *add* ◆

Fact:

The following equation holds between the specifications *OPT* and *NOPT*

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$\quad val(usedregb^{OPT}(sv[occ(e)])) = val(usedregb^{NOPT}(sv[occ(e)]))$ ◆

Proof

The proof is done using the following complete set of occurrence terms:
{ *sv*[*occ*(*natexpr*(*n*))], sv[occ(idexpr(id))], sv[occ(comp(e1, add, e2))],
sv[occ(comp(e1, sub, e2))], sv[occ(comp(e1, mult, e2))] }

***sv*[*occ*(*natexpr*(*n*))]:**

$val(usedregb^{OPT}(sv[occ(natexpr(n))])) =$
$\quad regval(usedregb^{OPT}(sv[occ(natexpr(n))]),$
$\qquad regIs(usedregb^{OPT}(sv[occ(natexpr(n))]), n)) = n =$
$\quad regval(usedregb^{NOPT}(sv[occ(natexpr(n))]),$
$\qquad regIs(usedregb^{NOPT}(sv[occ(natexpr(n))]), n)) =$

val(usedregb^{NOPT}(sv[occ(natexpr(n))]))

sv[occ(idexpr(id))]:
val(usedregb^{OPT}(sv[occ(idexpr(id))])) =
regval(usedregb^{OPT}(sv[occ(idexpr(id))]),
regIs(usedregb^{OPT}(sv[occ(idexpr(id))]), lookup(id, givenEnv))) =
lookup(id, givenEnv) =
regval(usedregb^{NOPT}(sv[occ(idexpr(id))]),
regIs(usedregb^{NOPT}(sv[occ(idexpr(id))]), lookup(id, givenEnv))) =
val(usedregb^{NOPT}(sv[occ(idexpr(id))]))

sv[occ(comp(e1, add, e2))]:
case analysis over
less(needed(sv[comp(occ(e1), add, e2)]), needed(sv[comp(e1, add, occ(e2))])) = false:

lhs:
val(usedregb^{OPT}(sv[occ(comp(e1, add, e2))])) =
regval(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
code^{OPT}(sv[occ(comp(e1, add, e2))])) =
regval(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
seq(code^{OPT}(sv[comp(occ(e1), add, e2)]),
seq(code^{OPT}(sv[comp(e1, add, occ(e2))]),
ADD(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
usedregb^{OPT}(sv[comp(occ(e1), add, e2)]),
usedregb^{OPT}(sv[comp(e1, add, occ(e2))]))))) =
regval(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]), code^{OPT}(sv[comp(occ(e1), add, e2)]))
& regval(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
code^{OPT}(sv[comp(e1, add, occ(e2))]))
& regval(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
ADD(usedregb^{OPT}(sv[occ(comp(e1, add, e2))]),
usedregb^{OPT}(sv[comp(occ(e1), add, e2)]),
usedregb^{OPT}(sv[comp(e1, add, occ(e2))]))
val(usedregb^{OPT}(sv[comp(occ(e1), add, e2)])) +
val(usedregb^{OPT}(sv[comp(e1, add, occ(e2))])) =
val(usedregb^{NOPT}(sv[comp(occ(e1), add, e2)])) +
val(usedregb^{NOPT}(sv[comp(e1, add, occ(e2))])) =

rhs:
val(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))])) =
regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),
code^{NOPT}(sv[occ(comp(e1, add, e2))])) =
regval(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),
seq(code^{NOPT}(sv[comp(occ(e1), add, e2)]),
seq(code^{NOPT}(sv[comp(e1, add, occ(e2))]),
ADD(usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),
usedregb^{NOPT}(sv[occ(comp(e1, add, e2))]),

$$\begin{array}{l}
\qquad\qquad\qquad\qquad \mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))]))))) = \\
\quad \mathit{regval}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{code}^{NOPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) \\
\qquad \& \ \mathit{regval}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad \mathit{code}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\qquad \& \ \mathit{regval}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad \mathit{ADD}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad\quad \mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad\quad \mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\quad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))])) + \\
\qquad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) = \\
\quad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) + \\
\qquad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))]))
\end{array}$$

***less*(*needed*(*sv*[*comp*(*occ*(*e*1), *add*, *e*2)]), *needed*(*sv*[*comp*(*e*1, *add*, *occ*(*e*2))])) = *true*:**

lhs:

$$\begin{array}{l}
\mathit{val}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))])) = \\
\quad \mathit{regval}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{code}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))])) = \\
\quad \mathit{regval}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{seq}(\mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))]), \\
\qquad\qquad \mathit{seq}(\mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]), \\
\qquad\qquad\quad \mathit{ADD}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad\quad \mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]), \\
\qquad\qquad\quad \mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))]))))) = \\
\quad \mathit{regval}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\qquad \& \ \mathit{regval}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad \mathit{code}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) \\
\qquad \& \ \mathit{regval}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad \mathit{ADD}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{occ}(\mathit{comp}(e1, \mathit{add}, e2))]), \\
\qquad\qquad\quad \mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)]), \\
\qquad\qquad\quad \mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) \\
\quad \mathit{val}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) + \\
\qquad \mathit{val}(\mathit{usedregb}^{OPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))])) = \\
\quad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(\mathit{occ}(e1), \mathit{add}, e2)])) + \\
\qquad \mathit{val}(\mathit{usedregb}^{NOPT}(\mathit{sv}[\mathit{comp}(e1, \mathit{add}, \mathit{occ}(e2))]))
\end{array}$$

rhs:

as in the other *needed* case.

***sv*[*occ*(*comp*(*e*1, *sub*, *e*2))], *sv*[*occ*(*comp*(*e*1, *mult*, *e*2))]:**

analogous to *add* ◆

Fact:

After performing the computation in register 1 the result of the expression is in *OPT* as well as in *NOPT.* ♦

Proof

Follows immediately from the fact:

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$val(usedregb^{OPT}(sv[occ(e)])) = val(usedregb^{NOPT}(sv[occ(e)]))$
$\quad and\ usedregb^{OPT}(occ(sv[e]) = 1 = usedregb^{NOPT}(occ(sv[e])$ ♦

Fact:

The following equation holds between the specifications *NOPT* and *SOURCE*

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$val(usedregb^{NOPT}(sv[occ(e)])) = value(sv[occ(e)])$ ♦

Proof

analogous to the fact

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$regval(usedregb^{OPT}(sv[occ(e)])) =$
$\quad regval(usedregb^{NOPT}(sv[occ(e)]), code^{NOPT}(sv[occ(e)]))$ ♦

Fact:

The following equation holds between the specifications *SOURCE* and *NOPT*

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$regval(1, code^{NOPT}(occ(sv[e])) = value(occ(sv[e]))$ ♦

Proof

Follows immediately from the fact:

$\forall\, sv_{Expr \rightarrow Expr}.\ \forall\, e_{Expr}.$
$val(usedregb^{NOPT}(sv[occ(e)])) = value(sv[occ(e)])$
$\quad and\ usedregb^{NOPT}(occ(sv[e]) = 1$ ♦

Fact:

It holds:

$OPT \rightsquigarrow_{\mathrm{beh}} NOPT$ ♦

Proof

Follows immediately from the above facts, since the only observable attribute is *value.* ♦

Index