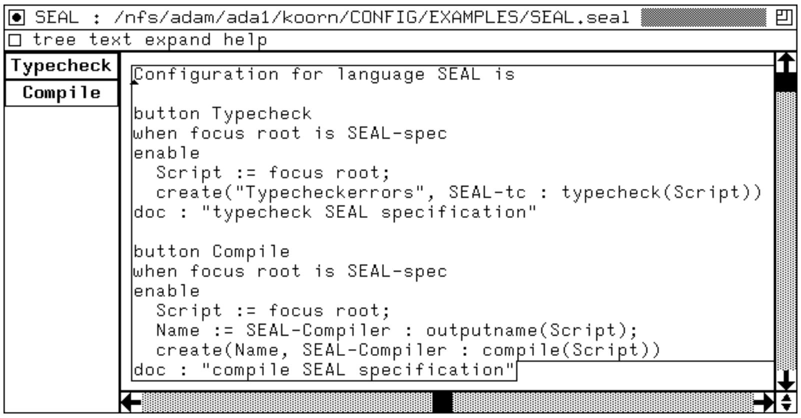
Generating uniform user-interfaces
for interactive programming environments



Jan Willem Cornelis Koorn

Generating uniform user-interfaces
for interactive programming environments

ILLC Dissertation Series 1994-2

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

Generating uniform user-interfaces for interactive programming environments

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr P.W.M. de Meijer
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op donderdag 3 februari 1994 te 13.30 uur

door **Jan Willem Cornelis Koorn**
geboren te Zandvoort

Promotor: prof. dr P. Klint.
Co-promotor: dr M. G. J. van den Brand.
Faculteit: Wiskunde en Informatica.

Printed by Febodruk Enschede.

Partial support for the research described in this thesis was received from the European Community under Esprit projects 348 (GIPE - Generation of Interactive Programming Environments) and 2177 (GIPE II - Generation of Interactive Programming Environments II).

ISBN: 90-74795-03-X

Contents

Acknowledgements

During the last five years I was employed as a researcher working in the GIPE project, the first year as an employee of B.S.O., and the last four as a PhD student of the University of Amsterdam. I received support from many people and I hereby express my gratitude to them.

First of all, I would like to thank Paul Klint, my promotor, and Mark van den Brand, my co-promotor. Paul has always been stimulating, friendly, in for any discussion no matter the subject, and helpful in many ways. Although Mark joined the GIPE team only recently, he has contributed a lot in the stage of writing this thesis.

Needless to say, I am very grateful to those who were willing to review my thesis: prof. dr J. A. Bergstra, prof. dr J. van den Bos, prof. dr K. van Hee, prof. dr G. Snelting, and prof. dr ir A. W. M. Smeulders.

Furthermore, the people in the GIPE team in Amsterdam have always been friendly and cooperative. Many thanks are therefore due to them all: Huub Bakker, Mark van den Brand, Hans van Dijk, Casper Dik, Dinesh, Job Ganzevoort, Jan Heering, Paul Hendriks, Jasper Kamperman, Paul Klint, Emma van der Meulen, Jan Rekers, Frank Tip, Susan Üsküdarlı, Emile Verschuren, Eelco Visser, Paul Vriend, and Pum Walters. Another nice experience was cooperating with GIPE researchers outside Amsterdam. Of all these people I would like to thank in particular Janet Bertot (INRIA Sophia-Antipolis) and Han Joosten (PTT research Groningen).

Then there are the people who contributed to a stimulating working environment: people of B.S.O., of C.W.I., and of the Programming Research Group of the University of Amsterdam. Thank you either for giving me the opportunity to do the job or for supplying help in whatever sense.

I have implemented a substantial amount of software and I received help from a number of people. Implementing the prototype of GSE (Chapter 2) was started by Monique Logger. She reported on her work in [Log88]. She was succeeded by Hans van Dijk with whom I worked for a full year. We reported on our work in [DK89] and [DK90]. Significant parts of the current version of GSE (Chapter 3) have been implemented by Huub Bakker and Paul Vriend. The three of us received help from Robert van Liere our X-windows, Motif, and networking guru. Claus Bo Nielsen helped implementing the Epoch client and server. Philippe Kaplan and Chris Love helped to "re-parent" the Epoch window.

Finally, thanks are due to Arie van Deursen and Han Joosten who permitted me to use parts of their SEAL scripts (Chapter 5).

Chapter 1

Introduction

1.1 Programming environment generators

Generating software, instead of writing it by hand, is nowadays a widely accepted technique and research in this field is blossoming. For example, there are a number of research projects aiming at the generation of programming environments from a formal definition of a (programming) language and a description of the desired user-interface.

A *programming environment* is a set of cooperating tools that help the programmer to carry out his or her task. They generally consist of an *editor*, used to construct a program, a *typechecker*, used to check whether or not the program complies with static semantic rules, and an *evaluator*, used to compute the dynamic semantics, i.e., "running" the program.

Systems generating programming environments use a *description* of these tools and a generator to obtain them. The model used by these systems is shown in Figure 1.1. Tools in a generated environment consist of parts which are common in all generated environments, and parts which are specific for the environment in question. Commonly used parts may either be generated or they may be linked to the generated environment in which case they must be generic. The advantage of using generic parts is that it saves generation time and it simplifies both the description and

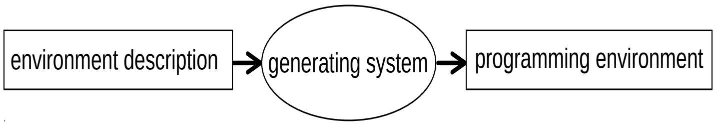


Figure 1.1: Model for generating programming environments

the generator. For this reason all programming environment generating systems use this technique whenever possible and a common approach is to use an editor as a generic part. However, only limited attention has been paid to applying generation techniques to obtain generic parts themselves. For editors this implies generating the graphical user-interface software and generating the editing facilities.

Most systems use *abstract syntax trees* [ASU86] as *internal* storage format for programs. The way programs can be entered and how visible the internal tree representation is, depends on the kind of editor used.

We distinguish three kinds of editors depending on the way a user interacts with them. If a user changes the text, which is then parsed by an external tool to derive the corresponding tree, the editor is said to be a *text* editor. Emacs [Sta81] is a typical example of a text editor. If a user changes the tree, which is then pretty-printed to derive the corresponding text, the editor is a *structure* editor. Emily [Han71] is the oldest structure editor. If the user is allowed to change either the text or the tree, the editor is said to be a *hybrid* editor. Several examples of this kind of editors are presented below. We will use the more general term *syntax-directed* editor for both structure editors and hybrid editors.

Systems generating programming environments all use generic *syntax-directed* editors in generated environments. This is a natural choice: a user interacting with such an editor manipulates the abstract syntax tree which is then immediately available for processing by a typechecker or an evaluator.

We will now briefly discuss several programming environment generating systems, their editors, and the facilities they offer to construct or adapt their user-interface.

1.1.1 Mentor and Centaur

One of the earliest research projects aiming at the generation of programming environments is Mentor [DGHKL80, DGHKL84, Lan85]. Mentor's successor Centaur [BCD+89] is part of the GIPE (Generation of Interactive Programming Environments) project [HKKL86]. Both systems generate environments featuring a structure editor but the user is also allowed to edit in a textual manner. The user selects a subtree in the structure editor and invokes a "text-edit" command. The selected tree is converted to a textual representation and is fed into Emacs, an existing text editor [Sta81]. After changing the text in Emacs, the user invokes a "parse" command which parses the changed text and replaces the selected subtree.

This scheme was already used in Mentor generated environments [Lan86]. Semantic processing is defined in Mentol (Mentor) and Typol (Centaur) [Kah87, Des88]. Extensions of the graphical user-interface of Centaur editors, for instance to connect a semantic processor to the editor, are written in LeLisp [LeL91], and are thus programmed by hand.

1.1.2 The Synthesizer Generator

Probably the most wide-spread system for generating programming environments is the successor of the Cornell Program Synthesizer [TR81], the Synthesizer Generator (SG) [RT89a, RT89b]. This system features hybrid editors in generated environments. All text editing features have been implemented separately, i.e., no use is made of an existing text editor. Switching from text editing to structure editing or vice versa is implicit and there may be more than one textual selection within the same editor. *Editing rules* define which language constructs may be edited in what *mode*, i.e., textually, structurally, or both. The current version, a commercial product exploited by GrammaTech Inc., uses an extension of an existing text editor, xedit, for text editing. Semantic processing is defined by attribute evaluation rules. SG editors have a graphical user-interface and the current version of SG features ways to connect semantic processors to the editor.

1.1.3 PSG

Yet another system generating programming environments is PSG (Programming System Generator) [BS86a, BS86b]. PSG generated editors are of the hybrid kind and both types of editing have been implemented separately[1]. Switching from structure mode to text mode is implicit, but the reverse is explicit. More than one textual selection within the same editor is allowed. An interesting feature of PSG editors is their filtering of menus from which structural editing commands are invoked. The filter prevents the presence of menu entries that will result in trees that are *semantically* incorrect. There is no facility to define semantic processors other than those used for static semantics and dynamic semantics. Therefore, PSG editors have a fixed graphical user-interface.

[1] [BS92] reports the use of an existing text editor, however, *which* one is not mentioned.

1.1.4 Gandalf

Editors generated with the ALOE generator of the Gandalf project [HN86] are structure editors and thus lack text editing facilities. However, recent experiments included text editing [NS90]. Another recent development in Gandalf is the *automated* customization of the editor's command set [Ler92]. For instance, the system learns, by monitoring user activity, certain frequently invoked combinations of commands. After the learning phase the combination is automatically executed after invoking its first command.

1.1.5 Other systems

Many other systems for generating programming environments exist of which we only mention a few. For instance, Pan [BGV92] features a hybrid editor in the PSG style. Mjølner/ORM [MBD$^+$90, Min90] editors are of the structure kind. The editor is equipped with a direct manipulation[Shn83] user-interface. Pregmatic [Bra92] editors are of the structure kind and use an existing text editor, xedit, in the Mentor/Centaur style. None of the systems mentioned feature extensible graphical user-interfaces based on software generation techniques.

1.1.6 The ASF+SDF Meta-environment

In this thesis we concentrate on user-interface and editing aspects of the ASF+SDF Meta-environment (Algebraic Specification Formalism plus Syntax Definition Formalism) [Kli93], another outcome of the GIPE project. The main distinctive feature of the ASF+SDF Meta-environment is the integration of the development environment and the generated environment into one interactive application. For example, the same editor is being used both for editing language definitions and for editing programs. An implication of this integration is the need for multiple editor instances. These editors, of the hybrid kind, are discussed in detail in Chapters 2, 3 and 6. Semantic processors are defined algebraically and are implemented using term-rewriting techniques. Each tool in the generated environment uses an abstract syntax tree —or *term*— as data format for both input and output.

1.2 User-interface definition tools

We already mentioned that projects aiming at the generation of programming environments have paid only limited attention to applying generation techniques to the graphical user-interface. However, others have concentrated on such techniques. Two approaches can be distinguished in this field: the *toolkit* approach and the *user-interface management system* approach. We now briefly discuss these methods and the implications of using them in our context.

1.2.1 User-interface toolkits

A user-interface *toolkit* can be characterized as a software library. It offers functions to create graphical objects, like `create-button`, but also functions to compose the layout of a window, like `column`. A menu, for instance, may then be programmed as a `column` of buttons. Many toolkits for building user-interfaces exist, such as Xt for X-windows [SG86] and Toolbox for the Macintosh [Che87].

Parts of a toolkit can also be *generated*, for instance, by using ESTEREL [BCG86, BC84] or Squeak [CP85]. Descriptions in one of these formalisms define the *behavior* of objects, i.e., their *reaction* to *events*. For example, an object changes its color —the reaction— when the mouse enters the object's window —the event—. Note that here too, the toolkit forms a combination of generated functions (for objects) and generic functions (for composing window layout).

1.2.2 User-interface management systems

While toolkit generation is only concerned with generating objects, methods for generating entire graphical user-interfaces have also been proposed. These methods lead to *user-interface management systems* [HH89, Hee92]. Such systems are not only used to define a graphical user-interface, but they also manage it at run-time, and they are used for "connecting" the user-interface to the application (non-user-interface) part of the software.

Several description techniques have been used, including *state transition diagrams*, see e.g. [Jac86], *grammars*, see e.g. [Bos88], *abstract events*, see e.g. [Hil86], and *graphics* in combination with *constraints*, see e.g. [MGD$^+$90]. In these descriptions one indicates which function to call when an event occurs. For example, the function `paste` is called when the button labeled `Paste` is pressed.

1.2.3 Using user-interface definition tools in our context

In the context of generating interactive programming environments these methods for user-interface definition can not be used without writing additional software. Toolkits require additional software for the layout of windows, user-interface management systems lack knowledge of the underlying application. The latter plays a role when dynamic changes in the user-interface, such as temporarily disabling a button, depend on the current data stored in the application. The point here is that currently existing user-interface management systems are designed as *general purpose* tools, they are not dedicated to systems generating programming environments.

1.3 Our goals

Obtaining *uniformity* of all user-interface aspects of the ASF+SDF Meta-environment is our primary goal. This immediately implies preventing a situation where users are confronted with yet another set of editing commands. Building an editor to be used in the ASF+SDF Meta-environment has two implications by itself. First, since multiple editor instances are used, the editor should be designed such that it can be used as a generic building block. Second, ASF+SDF specification writers may define *any* tool operating on a program's abstract syntax tree. These tools must somehow be "connected" to the editor, requiring an extensible user-interface and a "connection mechanism". Additional goals are: an efficient and easily maintainable implementation, and extensibility and customizability of all editing facilities. Summarizing, our goals are:

- ensuring the uniformity of all user-interface aspects;
- building an editor which can be used as a generic building block;
- incorporating an existing text editor to obtain first class text editing;
- introducing a mechanism to connect tools to the user-interface; and
- investigating the possibility to generate the editor itself.

1.3.1 Restrictions

The above formulated goals may be interpreted in a too broad sense. To make the subject of this thesis well-defined, we list our restrictions:

- All programming environments are generated by the ASF+SDF Meta-environment, except the Meta-environment itself.
- User-interfaces of generated programming environments consist of a collection of editors.
- Each editor is parameterized with a context-free language definition.
- Documents to be edited only contain *text*. We do, e.g., not incorporate pictures or sound.
- The mechanism to connect tools to the user-interface will use a textual description.
- Extensions of the user-interface of an editor will primarily be based on buttons and menus. These additional user-interface objects will be placed at predefined positions in the window of an editor.

1.3.2 Thesis overview

Chapter 2[2] discusses a model for smoothly integrating text and structure editing in the PSG style. It was used to implement a prototype of an editor of which the user-interface was built using the gfxobj toolkit [CI88]. Using the prototype as a generic building block leads to uniformity of all structure editing as well as uniformity of the graphical appearance and behavior of the user-interface.

In Chapter 3[3] we present how the text editing facilities of the prototype were replaced by Emacs, an existing text editor with rich text editing facilities. Furthermore, it discusses how we replaced the user-interface based on gfxobj by one based on OSF/Motif [Fou90]. We thus re-used the software forming the structure editing part of the prototype and replaced its two other parts (text editing and user-interface). This set-up has led to a *distributed* editor. Incorporation of Emacs leads to uniformity of text editing both inside as well as outside the ASF+SDF Meta-environment, it makes the software easier to maintain and it makes text editing extensible

[2]This chapter is a revised version of [Koo92]

[3]This chapter is a revised version of [KB93] and is joint work with H.C.N. Bakker

as well as customizable. Using OSF/Motif instead of gfxobj also promotes uniformity: it is widely used and many people are therefore familiar with the behavior of user-interfaces based on it.

Generating software necessary to connect tools to editors and to extend the editor's graphical user-interface is discussed in Chapter 4[4]. We present SEAL (Semantics-directed Environment Adaptation Language), a dedicated user-interface definition language for the ASF+SDF Meta-environment. It was designed such that tool connections can easily be established. The SEAL compiler and its run-time code ensure that no additional programming is required. This leads to uniformity of the user-interface extensions, and makes structure editing extensible. Furthermore, the compiler itself is completely written in ASF+SDF making it easy to maintain and easy to change or to extend. Finally, this approach is convenient for users familiar with the ASF+SDF Meta-environment, i.e., the tool writers, since they use the same system for writing tools as well as for writing user-interface definitions for their tools.

The SEAL language introduced in Chapter 4 can be used in a wider range of applications than only for connecting tools to an editor. For instance, it permits using a set of cooperating editor instances and it can be used to define user dialogues. Besides illustrating SEAL's potential power and presenting an overview of typical applications, Chapter 5 mainly serves as an assessment of SEAL's practical merits when defining user-interfaces.

In Chapter 3 we discussed how all text editing facilities of our editor were replaced by an existing text editor, and in Chapters 4 and 5 we discussed the generation of the editor's user-interface. The final step is generating its third and last component: the structure editing facilities. Chapter 6 discusses the feasibility of generating these from a description in ASF+SDF. We present a formal definition of a generic structure editor which forms a term. This term can be manipulated in a generated environment using SEAL to model the editor's commands. This chapter may be viewed as a step towards bootstrapping the ASF+SDF Meta-environment, but it also serves as a study of what structure editing exactly is.

Finally, we assess our results and state conclusions in Chapter 7.

[4]This chapter is a revised version of [Koo93]

Chapter 2

GSE: a generic syntax-directed editor

We present a syntax-directed editor in which all language dependent parts are parameterized. The editor provides operations on the text as entered by the user and does not depend on a pretty-printer for reconstructing the text from an internal tree structure. This approach has consequences for the data structure used, since both the text, as well as the corresponding tree, have to be stored and maintained. Also, a two way mapping between text and tree is needed. We present an elegant and efficient way to maintain these mappings. Furthermore, we discuss possible uses of the editor and the possibility to connect tools such as a typechecker or compiler. We conclude with summarizing the advantages and disadvantages of our approach.

2.1 Introduction

Our primary goal is to generate uniform user-interfaces for interactive programming environments, i.e., sets of cooperating tools that help the programmer to carry out his or her task. These generally consist of an editor, a parser, a type-checker and a code generator. In this chapter we concentrate on the role of the editor, the tool used to construct a program, within the environment. We describe the *prototype* version of the editor, its current version is described in Chapter 3.

We have developed a hybrid editor called GSE (Generic Syntax-directed Editor). Its role within the GIPE project is to serve as a building block for the ASF+SDF Meta-environment. GSE must therefore be generic, i.e., parameterized with a syntax definition and an optional set of semantic tools.

Creation of an instance of GSE for, say, Pascal, hence requires a definition of the Pascal syntax. Optionally, semantic tools like a typechecker, evaluator, or compiler can be defined and connected to the Pascal editor. Our goals in designing GSE were:

- smoothly integrating text and structure editing;
- being completely language-independent; and
- maximizing user freedom.

Chapter overview

In Section 2.2 we present a model for hybrid editing. This model leads to a situation in which text and structure editing are smoothly integrated —our first design goal— and language independency is achieved —our second design goal—. Next, we describe an important aspect in the implementation of a prototype of GSE in Section 2.3: mapping text-parts to subtrees and vice versa. Section 2.4 is concerned with our third design goal: maximizing user freedom. Achieving this goal leads to a refinement of the editing model. We briefly discuss connecting semantic tools to the GSE prototype in Section 2.5 and using GSE in the ASF+SDF Meta-environment is the subject of Section 2.6. Finally, we present conclusions in Section 2.7.

2.2 A model for hybrid editing

2.2.1 Why hybrid editing?

In the early eighties, there has been some debate whether structure editors should, or should not, abandon plain text editing facilities [Wat82, NHE$^+$83, Sha83]. A quite widespread objection to this hybrid approach comes from people advocating "pure" structure editing ("pure" in the sense of editing without any textual input, as in the Emily [Han71] editor) who argue that you do not need textual input. Surely this is true, but the argument can be reversed: text editors do not need commands for structured editing. People advocating "pure" structure editing sometimes argue that you do not need a parser. This is also true, but, as we will show in this chapter, this argument can be reversed as well: hybrid editors do not need a pretty-printer. The advantages of an editor based on parsing and not on pretty-printing, are:

- there is no need for a pretty-print definition;
- users do not have to customize the pretty-printer; and
- users have complete control over the layout of the text entered (this is particularly true for comments).

The disadvantages of an editor based on parsing and not on pretty-printing are:

- the text as well as the abstract syntax tree have to be stored in the editor's data structure; and
- there must be a two-way mapping between subtrees and text-parts.

In our view, the disadvantages are outweighted by the advantages, because:

- the need for maintaining two representations costs space, but yields speed when simple program constructs must be entered; and
- the mappings needed can be defined in an elegant and, as we will show in Section 2.3, efficient way.

Summarizing, we take the stand that all operations on a program are *textual changes*, which may have an update of its tree representation as side-effect.

2.2.2 GSE's editing model

The most important concept in GSE's editing model is the so called *focus* which designates a subtree. A focus is a pair [*text-part*, *tree*], where *text-part* is that part of the text that corresponds to the focus' *tree*. This tree is a subtree of the abstract syntax tree of the whole program and therefore corresponds to some language construct. Normal text editing (cutting, pasting, inserting characters, etc.) is *only* allowed inside the focus. A *cursor* indicates the current character position inside the text. All text outside the focus is guaranteed to be syntactically correct. Moving the focus to another part of the text is done by invoking a *navigation* command, like "go to the next child", "go to the previous child" or "go to the parent".

There is one navigation command which does not correspond to a tree traversal primitive, but to a breadth first search in the tree: the mousepoint command. It is invoked when the user points with a mouse device at some character and clicks. GSE interprets this as a request to move the focus to the smallest subtree S such that the text corresponding to S contains the character pointed at. Since the focus' tree is a subtree of the abstract syntax tree, the net effect of clicking on a part of a language construct is to move the focus to that construct and to move the cursor to the character pointed at.

2.2.3 Consequences of the editing model

GSE guarantees that all text outside the focus is syntactically correct, so, if the focus' text has been modified, any focus move requires parsing of the text inside the focus. If parsing of the focus' text succeeds, the focus' tree is replaced by the result of parsing. If not, it would be too restrictive to reject the focus move request, since this would force the user to correct a syntactical error before anything else can be done. A more user-friendly method is therefore adopted: *each* mousepoint command results in a cursor move to the character pointed at by enlarging the focus if parsing fails. In the latter case, the focus is moved to the smallest subtree S such that the text corresponding to S contains both the character pointed at as well as the old focus' *text-part*. By doing so, we obtain the "point-and-start-to-type" property found in most text processors.

Since GSE views *all* operations on a program as textual changes, we need some mechanism for editing incomplete programs. In our case, we require that every non-terminal of the grammar used has a unique textual representation which can be used as a placeholder. This requirement is, in contrast to e.g. SG (cf. Section 1.1.2), automatically fulfilled by the ASF+SDF Meta-environment. Placeholders correspond to special nodes in the tree representation. Positioning the focus at such a node results in placing all production rules corresponding to the non-terminal in a special menu. Upon selecting an entry in this menu, GSE replaces the focus' text by the textual representation of the production rule. For instance, a non-terminal named "`EXP`" is represented as `<EXP>`. If "`EXP ::= EXP + EXP`" is a production rule, a focus positioned at `<EXP>` might thus be replaced by `<EXP> + <EXP>`. Note that this approach also allows the textual insertion of non-terminals inside the focus.

2.2.4 Text mode and structure mode

In the previous section, we saw that parsing the focus' *text-part* is only necessary when a textual change occurred. This property is used to define the *mode* of the editor. The editor is in *text* mode when the focus' *text-part* has been changed since the last successful parse, otherwise it is in *structure* mode.

One of our requirements is to build an editor in which the "switch" between these two modes is implicit. This can be achieved as follows:

- When the editor is in structure mode, any textual change causes a switch to text mode.
- When the editor is in text mode, it stays in text mode after any textual change, or an unsuccessful parse.
- When the editor is in text mode, a successful parse causes a switch to structure mode.

2.3 Mapping between text-parts and subtrees

Our model for hybrid editing requires a two-way mapping between subtrees and text-parts. We will describe in this section how these mappings can be defined in an elegant and efficient way.

2.3.1 Maintaining positional information

The key idea in the design of the mapping between text and subtrees, is to store *position information* as an annotation in nodes of the abstract syntax tree. This has to be done by the parser. In principle, the editor needs the begin and end points of the text corresponding to each node. In this way, it can navigate through the tree and simply look up the begin and end point to implement a mapping from subtrees to text-parts.

Mousepointing can be viewed as a mapping from a text-part to a subtree: clicking on a character or selecting a part of the text identifies *two* points: the points before and after the selection. A breadth first search through the tree can be used to yield the smallest subtree whose corresponding text-part contains the selection.

However, with the above method, the position information stored in a large number of nodes must be updated if the text in the focus is changed. Consider the addition of a few lines in the focus. Then, all line numbers stored in nodes whose corresponding text-part lies after the focus (right neighbor nodes of the focus and their children, but also right neighbor nodes of the *parent* of the focus and their children etc.) must be updated. Also, the line numbers stored as the end point of the parent of the focus, and its grandparent etc. need to be updated. The issue of updating column

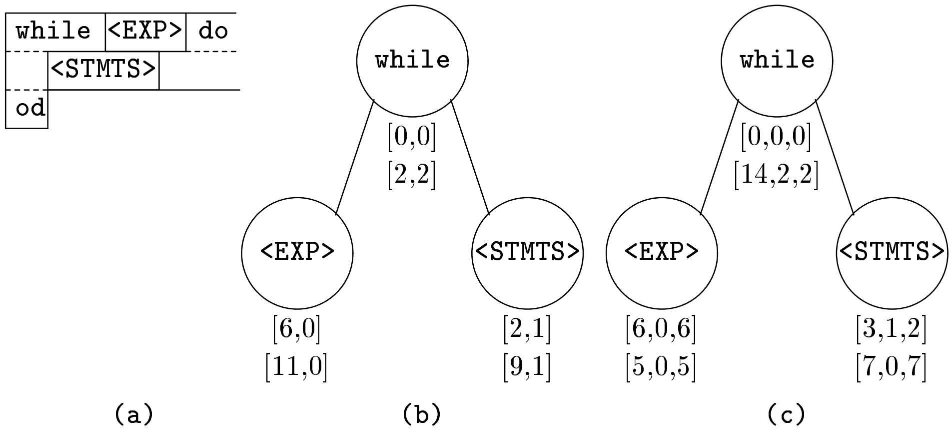


Figure 2.1: (a) division of text into text-parts, (b) annotation with points, and (c) annotation with size-and-shape structures.

numbers is even more complex. In the general case, textual positions must be updated in almost every node, resulting in an update function which is in time proportional to N, where N is the number of nodes in the tree. In the next section, we will show how we can obtain a $O(\log N)$ algorithm.

2.3.2 Size-and-Shape structures

We have developed a method which does not use begin and end points but so called *size-and-shape* structures to represent the information needed to compute the mapping between text and tree. These structures do not contain any positional information at all, only *sizes* are stored (substring lengths and the number of newlines). Size information is handled such that given one (begin or end) point of a text-part in size-and-shape format, the other point (and thus the *shape* of the text-part) can be calculated easily. Also, we can calculate a size-and-shape structure given two points and the text representation itself.

Consider the example in Figure 2.1. Nodes in the tree of Figure 2.1(b) are annotated with points, the begin point (topmost pair) and the end point (bottom pair). In Figure 2.1(c), nodes are annotated with two size-and-shape structures, the so called *shift-text* (topmost triple) and the so called *item-text* (bottom triple). The former describes the displacement of the begin point relative to some other point (e.g. the begin of the parent node, or the end of a neighbor node), the latter describes the displacement of the

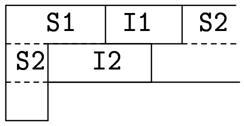


Figure 2.2: *Left-and-parent-relative* representation: division of an item-text into children's shift-texts and item-texts

end of the node relative to its begin. Each structure defines the text-part by three positive integers:

- the number of characters on the first line;
- the number of newline characters; and
- the number of characters on the last line.

Note that if there are no newlines in the text-part, the first integer equals the last and that newlines are *not* included in the character counts of the first and last line.

Using these size-and-shape structures, one can represent several *relative* forms of positional information. We will not discuss alternatives here, but present our choice, the *left-and-parent-relative* representation. In this representation, the "shift-text" describes the text-part formed by the *end* of the left neighbor of the node up to the *begin* of the node. The leftmost child's shift-text however describes the text-part formed by the *begin* of the parent up to the *begin* of the leftmost child. The "item-text" describes the text-part corresponding to the node itself. In Figure 2.2, we show the division of an item-text into its children's shift-texts and item-texts, corresponding to the case presented in Figure 2.1.

Some examples are in order. The "shift-text" of the <EXP> node ("S1" in Figure 2.2) is [6,0,6], because it is the leftmost child of the parent node and the text-part between the parent's begin and the node's begin contains no newlines and six characters (the word "while" plus a space). The "item-text" of the <EXP> node ("I1" in Figure 2.2) is [5,0,5], since this text-part is the word <EXP>, which consists of five characters. The "shift-text" of the <STMTS> node ("S2" in Figure 2.2) is [3,1,2], because the text-part between the end of its left neighbor's "item-text" and the node's begin contains three characters on the first line (a space and the word "do"), it contains one newline and finally, two characters on the last line (the two spaces forming the indentation).

Calculating the focus points for the <EXP> node is done as follows. We know that the while-statement begins at [0,0]. Next, we calculate the begin point of `<EXP>` by inspecting its "shift-text" annotation. This annotation, [6,0,6], does not contain newlines, so, `<EXP>` begins at column number $0 + 6 = 6$, and line number $0 + 0 = 0$. Since the begin of `<EXP>` is now known, we can calculate the end in the same way, yielding [11,0]. Given the end of `<EXP>`, we can calculate the begin of `<STMTS>` by inspecting its "shift-text" annotation: [3,1,2]. The line number is $0 + 1 = 1$, the column number is 2, because there are newlines in the "shift-text". The begin of `<STMTS>` is thus [2,1].

The main advantage of this representation is that annotations of the nodes whose corresponding text-part appears before or after the focus are *invariant* under editing. This property can be verified by observing that:

- we do not store positions but sizes;
- we do not allow editing outside the focus; and
- we use the left-and-parent-relative representation.

Again, we will give an example. Consider editing the `<EXP>` node, e.g. changing it to "`x`". Note that the shift-text part of `<STMTS>` is now displaced, but its size-and-shape representation is unchanged, since characters belonging to this part are not edited. Calculating the begin of `<STMTS>` uses the *new* end of the `<EXP>` node and the size-and-shape representation of the shift-text part of `<STMTS>`. Therefore, this will always yield the new begin of `<STMTS>`. A similar argument holds for nodes before the current focus. This leaves us in a situation where only the size-and-shape information of the parent nodes might need an update, which leads to a $O(\log N)$ algorithm.

2.4 Maximizing user freedom

The editing model presented so far imposes some undesirable restrictions on the user. For example, editing outside the focus is not allowed (cf. Section 2.2.2). In this section, we discuss our approach to alleviate this, and introduce a special treatment of list-items.

2.4.1 Generalizing commands and focus enlargement

The problem of not being allowed to edit outside the focus is solved by giving a very general meaning to editing commands in combination with automatically enlarging the focus whenever necessary and possible. Note that the need for automatic focus enlargement arizes when a user wants to *delete* a piece of text (partly) outside the focus, or, as we saw in Section 2.2.3, wants to move the cursor to a position outside the focus, while the focus text is syntactically incorrect.

We will illustrate this approach by presenting a typical example. Consider a focus somewhere in the text, and the cursor positioned at the begin point of that focus. If the user wants to delete the character just before the focus begin, he may invoke the "delete-char" command, in which case GSE reacts by enlarging the focus before the actual deletion of the character takes place. Of course, it is also possible that such a general interpretation of the command is impossible. In such a case, GSE does nothing. GSE thus reacts to text editing commands just like any other plain text editor does, i.e., the user is not hindered by the focus during text editing.

To summarize which action should be taken when the editor is in a certain mode, we present all actions for each mode in Table 2.1.

2.4.2 Treatment of lists: sublists and list-items

Unfortunately, automatic enlargement of the focus has as disadvantage that the focus tends to grow. As a consequence, the time needed to parse the focus will also increase. To alleviate this problem we allow the focus to be placed on a *sublist* as well, which results in a smaller focus in many cases. In fact, operations on list-items, sublists, and lists are essential for *any* structural editor, since these are very common language constructs (lists of definitions, statements, functions, etc.) and changing one item into several items, or deleting items is a very common editing operation. Besides focus enlargement, list related operations in GSE are:

- deleting one or more items from a list;
- changing one or more items in a list into new items; and
- adding an item to a list.

We will now discuss these operations in some detail.

First, consider deleting item i_k from a list in which items are separated, for instance by semi-colons. In this case we would end up in a situation where items i_{k-1} and i_{k+1} are separated by *two* semi-colons. When the language used does not allow empty items in the list, this implies a syntactical error. To prevent this, GSE also deletes the separator. Deleting a sublist of a list is handled in a similar manner.

Command	Editor in text mode	Editor in structure mode
cursor motion	If the new cursor position is outside the focus, compute a larger focus containing that position.	The new cursor position should be inside the focus and the focus should be as small as possible $\Rightarrow$ compute a new focus accordingly.
text insertion	The focus size increases $\Rightarrow$ increase the focus end.	Text mode required $\Rightarrow$ switch to text mode and perform text insertion.
text deletion	If the text to be deleted is not inside the focus, compute a larger focus containing it. The focus size decreases $\Rightarrow$ decrease the focus end.	Text mode required $\Rightarrow$ switch to text mode and perform text deletion.
parsing	Try to switch to structure mode by parsing the focus text. If parsing succeeds, switch to structure mode.	The focus text has already been parsed $\Rightarrow$ no action required.
navigation	Structure mode required $\Rightarrow$ try to switch to structure mode by parsing the focus text. If parsing succeeds, switch to structure mode and perform navigation.	Navigation implies a focus move $\Rightarrow$ compute the new focus. If the cursor is not inside the new focus, move the cursor such that it is.

Table 2.1: Overview of all actions for each mode.

Next, consider changing one item in a list to a sublist of items. In such a situation, reparsing of the focus' text results in an error, because the sort of the old focus tree (list-item) does not equal the sort of the new one (list). However, when the focus is enlarged to the parent node and then reparsed, we would *not* encounter this error. To prevent this, GSE could first enlarge the focus and parse it, but this would increase parse time. A different scheme is therefore adopted: GSE always parses a list-item in the focus (and also a sublist) *as if it were a list*. If this parse succeeds, GSE takes the resulting tree (a list of trees) and inserts its elements one by one in the parent list.

Finally, we allow insertion of new items in a list (or sublist) by an explicit command ("insert-hole") when the focus is positioned at a list-item, a sublist or a list. GSE knows two variants of this command: inserting *before* or *after* the focus. The inserted "hole" is a placeholder for a list-item. This command can be generalized as well: when the current focus is not a list-item, a sublist or a list, the tree is searched upwards for such a node and the "hole" is inserted at that place. In all these cases GSE adds a separator between list-items when necessary. Furthermore, the focus is moved to the new list-item. Finally, when the focus is at a list-item, a sublist or a list, which is still syntactically incorrect, GSE just pastes the textual representation of the "hole" before (or after) the focus.

2.5 Connecting tools

A common approach to connecting tools to an editor, like a type-checker, pretty-printer or interpreter, is to share the same underlying data representation of programs. Commonly, abstract-syntax trees are used [RT89a, DGHKL80, BS86a, Not85]. In GSE's case this is not different. Tools are connected to a GSE instance "at creation time" by placing additional entries in a special menu of the editor's user-interface. The function associated with each entry may ask the editor for its current tree, but might ask for any other part of GSE's data structure as well. This is a very simple approach to connecting tools, a more advanced method will be discussed in Chapter 4.

2.6 Current use of GSE

GSE, as described above, has been in use as part of ASF+SDF Meta-environment since 1989. In that system, instances of GSE are parameterized with a syntax-definition written in SDF (Syntax Definition Formalism, [HHKR89, Rek92]). SDF can be combined with a variety of semantic specification formalisms, but in the ASF+SDF Meta-environment it is combined with ASF (Algebraic Specification Formalism) [BHK89]. This combination forms a new specification formalism called ASF+SDF [HK89]. Tools, specified in ASF+SDF, can be connected to a GSE instance E when E is created from the Meta-environment. E is then customized by adding a menu entry to its user-interface. When this entry is activated, the Meta-environment asks E for its current tree, takes it as input for the tool and runs the tool[1]. Note that E is part of the *generated* environment. Therefore, the syntax used by E (as well as the tools connected to E) is defined by the ASF+SDF Meta-environment. Syntax definitions reside in GSE instances as well, it thus forms a "two level" editing process which is also addressed by [Kli93].

2.7 Discussion and conclusions

In this chapter we have shown that a smooth integration of text and structure editing can be achieved by adopting a textual approach. We have also shown that a generic syntax-directed editor can be built by parameterizing all its language dependent parts. Finally, we maximized user freedom by generalizing commands and by automatically enlarging the focus when necessary.

The advantages of our approach are:

- we do not need a pretty-printer;
- we obtain the same amount of user freedom as in plain text editors; and
- users can switch from text editing to structural editing, or the other way around, at any moment.

Removing the *need* for a pretty-printer does however *not* imply that pretty-printing has become impossible, since a pretty-printer, like other tools, can be connected to GSE.

[1] Recall from the previous section that a more advanced method will be discussed in Chapter 4.

The disadvantages of our approach are:

- the time needed to parse a focus is not constant, since it depends on the focus size; and
- during structure editing, the user has to add layout characters (spaces, newlines, etc.) manually.

A limitation of our approach is the incremental parsing technique used. Recall that *only* the text in the focus is reparsed. In some cases, this technique leads to an parsing error which would not occur if the complete text was parsed. In other cases, it leads to ill-formed trees. Tools processing such trees may therefore produce spurious results. Concluding, a proper incremental parsing algorithm should be incorporated. Such algorithms do exist, but their time or space requirements make them hard to use in an implementation. Furthermore, *multiple* focusses can then be introduced in many cases where currently the focus is automatically enlarged. This will improve performance as well since less text has to be parsed.

The introduction of multiple focusses would currently lead to more situations in which parsing errors occur while these are prevented if one, larger, focus is used. As an example, consider two syntactically incorrect focusses that are the children of an addition expression where the first contains the text "`(1`", and the second contains the text "`2)`". At the tree level corresponding to the expression the text reads "`(1 + 2)`", which is syntactically correct. However, due to the non-incremental parsing scheme used, the system would be unable to infer that joining the two incorrect focusses leads to the desired result. Note that in case of a single focus and automatic focus enlargement, this can not occur: after the creation of one of the incorrect focusses, any attempt to edit the second one leads to an enlargement that contains the erroneous part, i.e., the enlarged focus contains the text "`(1 + 2)`". We therefore prefer our scheme over a scheme that uses multiple focusses.

We have investigated the usefulness of a *substring* parser to overcome the limitation mentioned above [RK91]. Substring parsing appeared to be a too limited technique, and it is currently not used since it is less efficient. We omit further discussion here because we consider parsing as outside the scope of this thesis.

The prototype version of GSE described above forms our starting point for the generation of uniform, syntax-directed, user-interfaces for interactive programming environments.

Chapter 3

GSE and Emacs

We show that a hybrid text/structure editor can be built by combining an existing text editor with an existing structure editor. We describe how we have built such an editor using a client-server architecture for the communication between each of the three components involved (text editor, structure editor and user-interface). The main advantage of this technique is the re-use and integration of existing software components.

3.1 Introduction

In the previous chapter we presented a prototype of GSE, a hybrid editor with an implicit mode switch. Building such an editor is no easy job, as it is necessary to implement text editing facilities as well as structure editing facilities. Since both text editors and structure editors are already available, it is attractive to try to re-use existing editors, so that only the interconnection has to be coded. The subject of this chapter is therefore: how can we build a hybrid editor by combining an existing text editor and an existing structure editor?

We will try to integrate the structure editing facilities of the GSE prototype with the text editing facilities of Emacs [Sta81], an existing text editor. This exercise in software re-use has led to the current implementation of GSE: a distributed editor consisting of text editing facilities, structure editing facilities, and user-interface.

Chapter overview

In Section 3.2 we motivate our work and discuss work related to ours. In Section 3.3 we describe the general implementation model used, present some basic features of the text editor we used, and list the actual implementation in pseudo code. The user-interface component is the subject of Section 3.4. In Section 3.5 we show how multiple instances of our editor can be used as the user-interface of a programming environment. Section 3.6 discusses the use of the editor in the ASF+SDF Meta-environment, which is a programming environment generator. An assessment of how much code had to be changed, could be re-used, or had to be added is presented in Section 3.7. The chapter ends with Section 3.8, where we discuss our approach and make some final remarks.

3.2 Connecting two editors

3.2.1 Motivation

Our experience with the use of the GSE prototype as a generic building block in the ASF+SDF Meta-environment forms the motivation of the work presented here. The prototype versions of GSE were all implemented as new, stand-alone, systems. Therefore, they contained functions to manipulate text and to display text in a window. This approach has three disadvantages: it is a lot of work to maintain the code, text editing functionality is far less compared to what is provided in existing editors like, for instance, Emacs and new users have to learn new editor commands. These observations led to the idea to eliminate all text editing and displaying functionality from the GSE prototype and replace it by an existing text editor.

3.2.2 Re-using the model for hybrid editing

GSE's editing model was discussed in Section 2.2.2, extended in Section 2.4, and summarized in Table 2.1. There are a number of consequences of this model which have to be taken into account when we try to combine GSE with an existing text editor. First, since text editing is only allowed inside the focus, the cursor should always be positioned inside the focus. Note that when the cursor is moved by the user, this implies either parsing the focus' *text-part* or enlarging the focus such that it contains the cursor position. Second, again because text editing is only allowed inside the focus, deleting a piece of text that is outside, or partly outside, the focus also requires

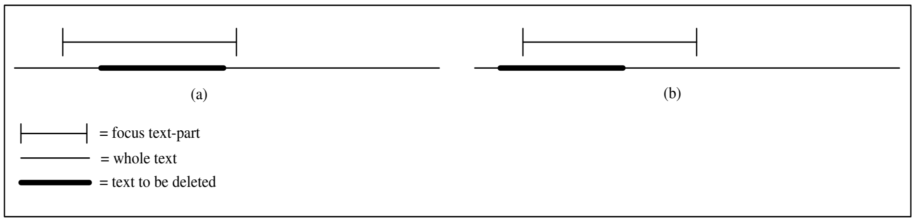


Figure 3.1: Deleting text may require focus enlargement

focus enlargement. However, deleting a piece of text that is inside the focus does not require a focus enlargement. Both situations for deleting text are shown in Figure 3.1. Finally, parsing the focus' *text-part* is, of course, only necessary when it has been changed since the last successful parse.

From the discussion above we conclude that keeping track of textual changes and of cursor movements is an essential part of GSE's editing model. Therefore, we are only able to use a text editor which offers these tracking capabilities. Epoch [Epo92], an extended version of Emacs, fulfills this requirement and we will use it as text editor.

3.2.3 Related work

The idea to extend a structure editor with text editing functionality, where an existing text editor is used, is not new. For instance, Lang reports in [Lan86] that at some stage, the Mentor system[DGHKL84] had a link with Emacs. This allowed users to select a subtree in the Mentor editor, edit its corresponding text in Emacs, parse the changed text and replace the subtree by the parse result. However, the approach used did not incorporate automated focus enlargement.

The successor of the Mentor editor, called ctedit[BCD$^+$88] currently uses Epoch instead of Emacs, but automated focus enlargement lacks here too.

Other hybrid editing systems sometimes have rich text editing functionality and customizable graphics, but use their own text editing and display facilities. For instance, Pan [BGV92] is extensible and customizable in the Emacs style and features completely unrestricted text editing. Another hybrid editing system with unrestricted text editing is PSG [BS92].

3.3 Implementation

Connecting an existing text editor with an existing structure editor, requires that adaptations are made to the source code of both editors. For GSE, this is not a problem, since we have implemented it ourselves, but Epoch was created by others. However, Epoch is extensible because it contains a Lisp interpreter. This allows us to extend Epoch with code necessary for connecting it with GSE. Here, we first describe the general implementation model we use, after which we discuss two essential Epoch features called *zones* and *hooks*. Next, we take a closer look at focus enlargement and focus moves. At the end of this section, we present our actual implementation.

3.3.1 General implementation model

The basis for our implementation is a *client-server* architecture, as found in, e.g., the X-window system [SG86]. In this architecture, components can run on different machines while the system as a whole appears to the user to be running on a single machine. The actual machines involved communicate with each other through a network. The client-server architecture enforces a strict separation between components: they can only exchange information by using the interface layer to the network software. Therefore, the implementation as a whole tends to be better maintainable.

Each component involved is a *server* to the others, which are the *clients*. If a client needs data stored in a server, it builds a *command* that contains, amongst others, the destination. The command is handed over to the network software which dispatches it to the appropriate server. Next, the server interprets the command which leads to calling a function in the associated component. Finally, the result of the function call is sent back to the client which interprets the answer. The general architecture is shown in Figure 3.2.

3.3.2 Implementing the focus using Epoch

Two features of Epoch are essential for implementing the focus concept: *zones* and *hooks*.

Zero or more *zones* may be associated with an Epoch text. A zone in Epoch is a data structure associated with a part of the text. It contains a *style*, a *start position*, an *end position*, and a *data* field[1]. A style describes

[1] An Epoch zone contains even more fields, but those are not relevant here.

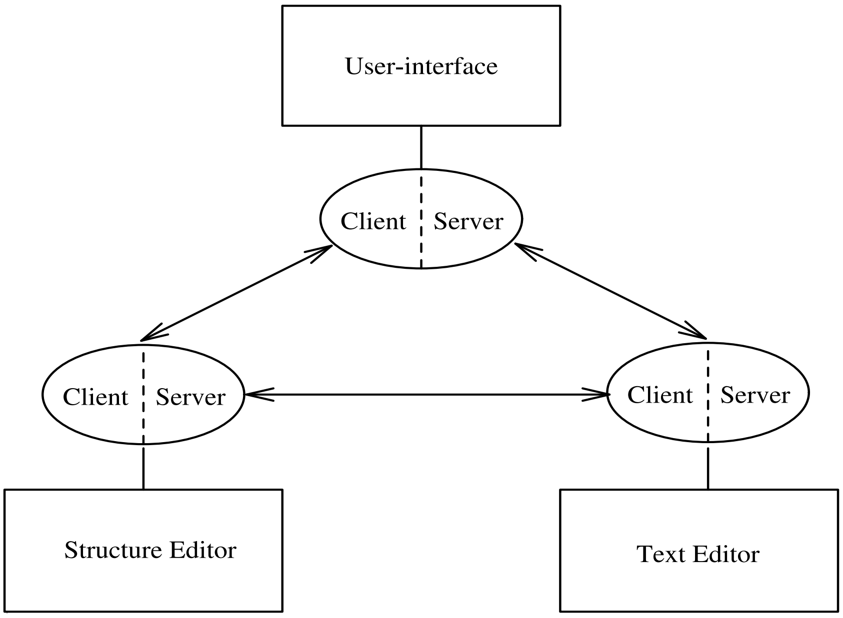


Figure 3.2: Client-server implementation model

how to display a zone, i.e., it contains font and color information. We will return to styles in Section 3.4. If the user inserts or deletes text, Epoch updates all data in its zones. For instance, if the cursor is positioned within a zone and a user inserts text in it, the zone's end position is updated. Since we need to keep track of all text in the focus by maintaining the focus start and end position, an Epoch zone is the natural choice for the data structure to implement the focus. We use the data field of a zone to mark it as "added by GSE".

A *hook* in Epoch is a variable which may hold the name of a function. If a function name is stored in a hook, Epoch calls the function with that name with predefined arguments. Hooks are mainly used to keep track of what is happening inside Epoch. For example, there is a hook called `after-movement-function` to which a function with one argument may be attached. Whenever the cursor is moved, the attached function is called with the old cursor position as an argument. We use this hook to check whether or not the cursor is still positioned inside the focus. For this check we need, of course, the new cursor position as well, which is not an argument of the function called, but we can obtain it directly from Epoch by calling an internal function.

Another hook is used to keep track of textual changes, since deleting text may lead to a new, larger, focus. Also, if the editor is in structure mode, we must record a mode switch when the text changes. We use the `before-change-function` for this purpose. The function associated with this hook is called before each textual change and is supplied with two arguments: the position of the change and the position of the end of the region affected. If the two positions are equal, the change is an insertion.

3.3.3 A closer look at focus enlargement and focus movement

Recall that the focus corresponds to a pair [*text-part*, *tree*]. Now consider the problem of computing a focus enlargement. First of all, any focus enlargement implies changing the focus' *tree* to its parent tree, or to any of its grandparents. Secondly, the new focus' end position depends on the old focus' end position, since there might be text inserted or deleted. Therefore, the new focus' end position can not be computed in GSE without asking Epoch for the old focus' end position. Furthermore, focus enlargement is sometimes necessary for deletions (cf. Figure 3.1). This implies that for some deletions Epoch needs to call a function in GSE, while for other deletions there is no such need. As a consequence, the response time of a deletion operation is not constant, which may be annoying for a user.

A different scheme is therefore adopted: for each focus surrounding the current one we add a separate zone to the Epoch text. This scheme is shown in Figure 3.3. New focus end positions are now immediately available, since Epoch updates its zones during text editing. If a focus enlargement is needed, the zone corresponding to the focus is deleted. The new focus zone is now the smallest of the remaining zones. In this way, we do not need any communication with GSE during text editing, yielding constant response times for deletions. However, GSE looses track of the focus' *tree* when the focus is enlarged. Therefore, when a parse command is invoked, which is a function in GSE, GSE first asks Epoch how many zones are associated with its text. By comparing this number with the number of focus parent nodes in the tree, GSE is able to compute how many focus zones were deleted since the last parse. Consequently, GSE moves the focus' *tree* to the computed parent node before the focus' *text-part* is parsed.

When the editor is in structure mode and the user moves the cursor, we might have to change the focus as well, since the focus' tree must be the smallest subtree S such that the text corresponding to S contains the character at the cursor position. In the new scheme this requires delet-

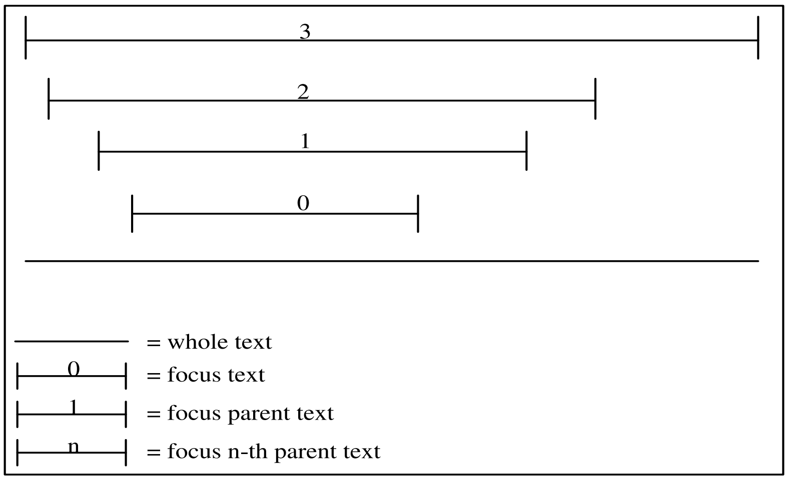


Figure 3.3: Focus zone, its parent zone, etc. for a focus with three parents.

ing the zones not containing the cursor position, and possibly adding new ones. In this situation, we call the `GSE-add-focus-zones` function, which is supplied with the number of remaining zones and the new cursor position. The smallest subtree S is computed in GSE and the focus' tree is set to S. Furthermore, GSE counts the number of nodes from S to the root and compares it with the number of remaining zones. If these numbers are unequal, GSE instructs Epoch to add a zone for S, the parent of S, etc., until the number of zones in Epoch equals the number of nodes from S to the root. Moving the cursor in structure mode is shown in Figure 3.4. Figure 3.4(a) shows the situation just before the move and Figure 3.4(b) shows the situation just after the move.

3.3.4 Actual implementation

We present our implementation as a list of functions in pseudo code. Functions in GSE have names starting with "`GSE-`", likewise, functions in Epoch start with "`Epoch-`". Epoch functions handling cursor moves and textual changes are:

```
Epoch-after-movement-function
  delete-zones-not-containing-new-cursor-position
  if in-structure-mode then
    call GSE-add-focus-zones (number-of-zones, new-cursor-position)
```

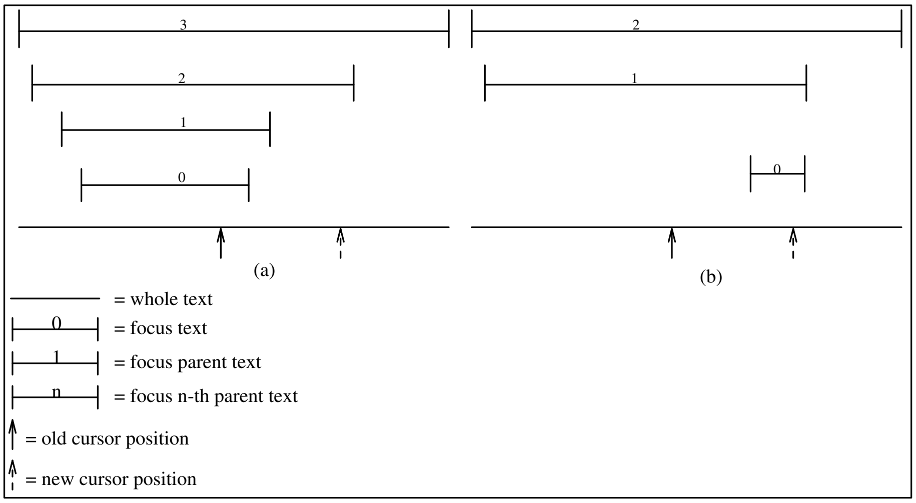


Figure 3.4: Focus zones (a) before and (b) after moving the cursor in structure mode.

```
Epoch-before-change-function
  if in-structure-mode then
    switch-to-text-mode
  if change-is-a-deletion then
    delete-zones-not-containing-entire-deletion
```

GSE functions handling parse and navigation commands[2] are:

```
GSE-parse
  if call Epoch-in-text-mode then
    N := call Epoch-get-number-of-zones
    Text := call Epoch-get-focus-text
    parse (N, Text)
    if parse-succeeded then
      call Epoch-switch-to-structure-mode
GSE-go-to-parent
  call Epoch-delete-smallest-zonez
GSE-go-to-next-child
  if call Epoch-in-structure-mode then
    if next-child-exists then
      new-focus-position := compute-position-of-next-child
      call Epoch-move-focus-zone (new-focus-position)
```

[2]We omit `go-to-first-child` and `go-to-previous-child` since these are similar to `go-to-next-child`.

3.4 The User-interface

There are two points in the editor under construction where the user-interface (UI) plays an essential role. First, the focus should be visible in the Epoch text. Second, the window in which the text is displayed must be "decorated" with a menu bar from which GSE commands, such as structure editing, parsing, and navigation commands, may be invoked. We discuss these aspects below.

3.4.1 Displaying the focus

There are two reasons why the focus should be visible in the text. First, if the user invokes a parse command, parsing may either succeed or fail. Clearly some form of feedback is required here. Second, if the editor is in structure mode, all structural editing commands are relative to the focus' *tree*. In this case, a user wants to know what the corresponding text part is before he or she decides which structural editing command is to be invoked.

We have chosen to implement this using Epoch's zone styles (cf. Section 3.3.2). We use three different styles for the focus zone: the *ok-style* when the editor is in structure mode; the *error-style* when parsing failed; and the *text-style* when the editor is in text mode. Parent zones are displayed in the text-style. All three styles use the same font, but different colors for the text and the text background.

3.4.2 Decorating the text window

Structure editing commands, such as replacing a placeholder by a template, are menu driven. Consequently, we have to add a menubar to the window in which the text is displayed. Note that this window is created by Epoch. We are therefore confronted with a situation where we want Epoch's window to be a *subwindow* of a window that contains the menubar. However, this can be handled by supplying a so called *parent-window* to Epoch's "create-window" function. For the implementation of windows that contain a menubar, we have chosen OSF/Motif™ [Fou90], a system for building graphical UIs based on the X window system [SG86].

3.4.3 Handling callbacks

It is standard practice in current windowing systems that each UI object —such as a button— has an associated *callback*. This is a function to be called when the UI object is activated by the user. In our case, we are unable to invoke such a function directly, because it might be a function in the text editor or in the structure editor component. These functions might require arguments only present in that component. Therefore, we need a more general mechanism to handle callbacks.

Recall from Section 3.3.1 that the UI is a separate component of our system. The creation of any window is thus the result of the interpretation of a command. All commands involving windows are invoked by GSE. The `create-window` command is supplied with a filename, the UI component creates an Epoch window displaying the file, and its surrounding window. This surrounding window has an *empty* menubar. The `create-window` command returns a *window identifier* to GSE. GSE stores this identifier in its data structure as a *shadow window*. Next, GSE invokes commands to add entries to pulldown menus. One command per entry. Each of these commands return an *object identifier*, which is stored in GSE's shadow window. We call this a *shadow object*. The GSE function to call when there is a callback from an object, is now stored in the shadow object.

A callback is now handled as follows. Consider a callback from an object with identifier I located in a window with identifier W. The callback from the UI object is first sent to the UI component, which builds a command indicating that there was a callback from I in W. Then, the UI component sends this command to GSE using the network. GSE searches the shadow object with identifier I in its shadow window with identifier W and calls the associated function, supplying it with the appropriate arguments when necessary.

If we want to call a text editor function from a UI object, this function is invoked by first creating a command at the GSE level and then sending it to the text editor component. At first glance, it seems that this scheme implies that there is no need for communication between the UI and the text editor component. However, these components do need to communicate, because the Epoch window must be created as a subwindow of our editor. Furthermore, a user might resize or destroy the editor window, in which case the Epoch subwindow must be resized or destroyed as well.

3.5 Using the editor in a programming environment

As with the older versions of GSE, the new editor is used in programming environments generated with the ASF+SDF Meta-environment. Generated environments are based on a *collection* of syntax-directed editors, and each editor may use a different language. Until now, we have discussed a situation in which there was only one combined editor, the question therefore is how to implement multiple instances.

3.5.1 Multiple editors

Epoch is able to handle several texts at the same time by displaying each text in a separate window. These Epoch windows are subwindows of OSF/Motif windows managed by the UI component. By replacing the structure editing component by a *GSE manager* (GM), we are able to implement multiple instances. The collection of syntax-directed editors, needed in a generated programming environment, is thus modeled by GM. We have to make some adaptations however, since in the case of multiple instances, the problem of how to identify each instance arises.

3.5.2 Identifying instances

The programming environment identifies each editor instance by means of a *filename*. For example, adding an instance to the environment is done as follows. After the user has supplied a filename, GM first creates a new GSE data structure in which the filename is stored. Next, a new window is created of which the identifier and the identifiers of UI objects are also stored in the new GSE data structure.

If a structure editor needs information stored in its corresponding text editor, the structure editor sends a command to the text editor component of the system. This command is supplied with a filename by which the text editor component can identify the appropriate text editor.

Callbacks are now handled by GM instead of by GSE itself. Each callback comes with a window identifier W and an object identifier I. GM responds to a callback by searching the shadow window with window identifier W in all GSE instances. When this shadow window is found, the object with identifier I is searched, and the associated function is called.

3.6 Using the editor in a programming environment generator

So far, we have only discussed the windows of our new editor itself. Besides these type of windows, there are other types of windows, such as: dialog boxes, error windows, etc. Furthermore, in the ASF+SDF Meta-environment (cf. Section 3.2.1), the notion of a *module-editor* exists. A module-editor is a composition of *two* editor instances which are displayed in one window. In an ASF+SDF module-editor, a user defines syntax rules in one of the editors, which can immediately be used in the second editor. This two level editing process is described in [Kli93]. Finally, in the ASF+SDF Meta-environment there are components that use editor instances, or other window based objects, as part of their user-interface. We now describe how we extended the implementation of the UI to allow the handling of an arbitrary number of additional components.

3.6.1 Window types

Windows, such as dialog boxes, error windows, etc., all have a *type*. Each window type has specific commands to build its window and has its own callback. For example, a dialog box with "OK" and "CANCEL" buttons is built using three commands: `create-dialog`, `add-button("OK")` and `add-button("CANCEL")`. Each command returns identifiers as before, such that a shadow dialog box can be built. Executing a callback from a button in a dialog, called a *dialog-callback*, now amounts to looking for a shadow dialog with the appropriate dialog identifier, searching the button in the shadow dialog and calling the associated function.

3.6.2 Combining editors in frames

Combined editors, such as ASF+SDF module-editors, are displayed in a window that contains subwindows, one for each editor involved. We call such a subwindow a *pane* and we call the surrounding window a *frame*. This window type is implemented in a similar manner as the editor window described earlier. That is, we send several commands from GM to the UI to build a combined editor. First, a frame is built that does not contain any panes. Second, we add a pane for each editor involved, which is an editor window with an empty menubar. Finally, the menubar of each pane is filled with menu entries.

Callbacks from objects in a frame now contain three identifiers: the frame identifier (F), the pane identifier (P) and the object identifier (I). In this way, we are able to use the same scheme for callbacks as described in Section 3.5.2, except that we have to search for a frame with identifier F first. This approach allows the addition of UI objects to a frame as well. A typical example of such an object is the `destroy window` button.

3.6.3 Windows created by other components

Besides the windows created by GSE, other components of the system may create windows as well. Consider, for example, a dialog box that is created by another component. Any callback from this dialog box will be sent to GM, which searches in all GSE instances for a shadow dialog with the identifier derived from the callback. This dialog will not be found, because the shadow dialog is not a part of any GSE instance. To solve this problem, we add a level of indirection: the *name* of the component that created the window. Any callback is supplied with this name as an argument. The system now becomes responsible for handling all callbacks. It first strips off the name argument, and then calls the callback manager for that component, this time without the component name. Consider, for instance, a callback from a frame that was created by GSE. In the system, the callback arrives as `frame-callback("GSE" F P I)`. The system maps this call to a call in GM: `GSE-frame-callback(F P I)`. Callbacks from frames created by the debugger (DB) however, arrive as `frame-callback("DB" F P I)` and are mapped to `DB-frame-callback(F P I)`. In this way, any component in the system can build its own user-interface and is by itself responsible for handling any callbacks. When a new component is added to the system it can use the library of user-interface functions immediately due to the fixed mapping of callbacks by the system.

3.7 Assessment: quantification of code re-use

How much effort was involved in changing GSE, implementing the network communication, and changing the user-interface? What was gained?

In this section we present how much code was involved when we made our new hybrid editor. We quantify the amount of code by counting lines of source code. While counting the number of source lines we included empty lines, comments and code used for testing and debugging. We sometimes list a total of the number of source lines used by a component. These totals should be interpreted with care since we add counts of sources that are written in different programming languages. That is, the old GSE is completely written in LeLisp [LeL91], Epoch is written partly in C [KR78] and partly in Emacs-Lisp [LLG90, Epo92], the new UI is written in C and the network communication is also written in C. For this reason we list the number of lines written in each language where we use the abbreviations `LL` for LeLisp, `EL` for Emacs-Lisp and `C` for C. We included `*.h` files while

Used for	old(LL)	new(total)	new(EL)	new(C)	new(LL)
Text functions	2178	2369	1478	-	891
Displaying text	3797	-	-	-	-
Building the UI	4971	6251	-	3158	3093
Network layer	-	4561	586	3924	51
Total (changes)	10946	13181	2064	7082	4035
Re-used	6331	6331	-	-	6331
TOTAL	17277	19512	2064	7082	10366

Table 3.1: Overview of sizes of the source code used by each editor version.

Part	total	EL	C	LL
Interface to rest of GSE	325	-	-	325
Message building and interpreting	551	185	-	366
Reactions to messages	1493	1293	-	200
TOTAL	2369	1478	-	891

Table 3.2: Breakdown of the size of the new text editing functionality.

counting C source lines, but we excluded `*.h` files used for hardware specific compilation.

Two major parts of GSE have been changed: the text editing component and the UI component. The third major part of GSE, used for parsing text and structure editing, is re-used. The sizes of the sources involved are listed in Table 3.1. We excluded the parser and the pretty-printer code, a total of 21348 lines, from the count of the re-used part. The numbers presented raise some questions. For instance, "Why is the new text editing component *larger* than the old one? (2369 new lines compared to 2178 old ones)". Indeed, at first glance this seems strange since we now use an existing text editor which is an *external* component. The answer is that in the new situation we have to interface with the network layer which implies some overhead. The overhead consists of building a command string, interpreting the command, building an answer string and interpreting the answer. To give an overview of how much code is used for the overhead and how much code is "really" used, see Table 3.2. This table shows that 1493 lines are "really" used. A similar effect can be found in the new UI. Here we wrote 6251 lines of new code and eliminated 4971 lines of code. Of the new 6251 lines, 5086 (6251 - 1165) are "really" used as shown in Table 3.3.

Part	total	EL	C	LL
Interface for building objects	2253	-	-	2253
Message building and interpreting	1165	-	731	434
Reactions to messages	2833	-	2427	406
TOTAL	6251	-	3158	3093

Table 3.3: Breakdown of the size of the new UI.

Component	LL	source	total	EL	C	LL	source
UI library	7663	`ext`	6348	-	6348	-	`int`
Epoch	-	-	208951	64720	144231	-	`ext`

Table 3.4: Overview of sizes of the external code used by each editor version. Code written by us is marked `int`, code written by others is marked `ext`.

Another question triggered by Table 3.1 is: "Why did you write 13181 lines of new source code in order to eliminate *less* code (10946 lines)?" We did this because we now have incorporated Epoch and this amounts to a gain in text editing functionality with a size of 208951 lines of code. For an overview of the amount of external source code used by each version we refer to Table 3.4. The main advantage of the new situation is the percentage of the code that has to be maintained in the future. This is reflected in Table 3.5. Also, we have created our new editor in such a way that different components can be run on different machines. The amount of source code involved in network communication is 4561 lines (cf. Table 3.1). This part is re-usable and may therefore be re-used by other parts of the ASF+SDF Meta-environment in the future.

	Old			New		
	`int`	`ext`	%	`int`	`ext`	%
GSE	17277	-	100	19512	-	100
UI library	-	7663	0	6348	-	100
Epoch	-	-	0	-	208951	0
TOTAL	17277	7663	69.3	25860	208951	11.0

Table 3.5: Overview of sizes of all source code used by each editor version. Code written by us is marked `int`, code written by others is marked `ext`.

3.8 Discussion and conclusions

We have shown that a hybrid editor can be built by connecting an existing text editor with an existing structure editor. The general advantages of this approach are:

- The text editing functionality of the hybrid editor is as rich as the functionality of the text editor.
- Users familiar with the text editor can use the hybrid editor immediately.
- Future improvements to the text editor are inherited.
- There is no need for maintaining the source code of the text editor by the implementors of the hybrid editor (and vice versa).

The advantages of connecting Epoch and GSE in particular are:

- We inherit the extensibility and customizability from Epoch.
- Each component of the hybrid editor may run on a different machine.

We are able to connect Epoch and GSE because Epoch offers some special functionality. These features are: extensibility, which we use to implement network communication; hooks, to keep track of cursor moves and textual changes; zones, to display and implement the focus concept; and windows that can be subwindows, to decorate a surrounding window with a menubar containing editing commands.

We have built our editor by extending Epoch and by removing all text editing and display functionality from GSE. In other words, we stripped the structure editor and extended the text editor. This leads to the question if a complementary approach would also work: can we strip an existing text editor and extend an existing structure editor. We expect that this will be much harder because in this situation we need to extend the display functionality of the structure editor for displaying unstructured text as entered during editing. Therefore, using the display functionality of the text editor seems the natural choice.

A related question is whether or not it would have been a better approach to extend Epoch with structure editing functions. However, this would require re-implementing GSE's structure editing functions as well as the parser and the pretty-printer. This amounts to re-implementing 6331 lines of GSE code (cf. Table 3.1) and 21348 lines for the parser and pretty-printer. If we compare the total of 27679 lines to the 13181 lines we now wrote, the approach chosen seems to be the best.

Chapter 4

SEAL: a semantics-directed environment adaptation language

The problem of how to connect a function, offered by a computational component, to a graphical user-interface part —such as a button— is addressed in a general way. We propose a solution that achieves a complete separation between the computational component and the user-interface component. We show that our approach can be used in the ASF+SDF Meta-environment and present a user-interface definition language for it, called SEAL. Code generated from a SEAL script can be combined with run-time code thus forming a User-Interface Management System. As a result, functions representing semantic tools, such as typecheckers, compilers, and program transformers, can be "connected" to the user-interface of a syntax-directed editor. Several examples of typical man-machine dialogues are presented and the suggested approach is compared with other techniques.

4.1 Introduction

Our primary goal is ensuring the uniformity of all user-interface aspects of the ASF+SDF Meta-environment. The user-interface of this system consists of a collection of editor instances which were discussed in detail in the previous chapters. In this chapter we concentrate on the question of how computational tools, such as typecheckers, interpreters, and compilers, can be connected to a uniform, syntax-directed, editing interface.

We suggest an approach based on a distinction between so called state inspection functions and state manipulation functions. Although our approach is dedicated to the ASF+SDF Meta-environment, we believe that it can be used in other application domains as well. We therefore discuss the subject in a more general context, that of *User-Interface Management Systems* (UIMS) [HH89, Hee92]. These are systems used for the development of the user-interface component of an interactive application.

Chapter overview

Section 4.2 introduces UIMSs and discusses our approach to designing a UIMS dedicated to the ASF+SDF Meta-environment. In Section 4.3 we give a number of examples, each illustrating a typical problem which a UIMS has to solve. In Section 4.4 we describe our application domain in an abstract way. From this description we derive which functions are state inspection functions and which functions are state manipulation functions. We continue with presenting a UI definition language called SEAL (Semantics-directed Environment Adaptation Language) in Section 4.5. Scripts written in this language can be used in combination with functions in a computational component generated from an ASF+SDF specification. An introduction to the ASF+SDF formalism is therefore included. We also present all language constructs and predefined functions of SEAL. This section is concluded with a number of example scripts corresponding to examples given in Section 4.3. Next, we briefly describe related work in Section 4.6. Finally, we summarize our approach, list its advantages and disadvantages, and make some final remarks in Section 4.7.

4.2 User-Interface Management Systems

In the development process of software for interactive applications, one can distinguish two subprocesses:

- development of a computational component (the actual application program); and
- development of a user-interface component.

The advantages of *separate* development, called *dialogue independence* in [HH89][1], of the two components are widely recognized. Clearly, at a certain

[1] We adopt the terminology of this paper.

moment in time both components have to be "connected". Commonly a *Programming Environment*, in combination with a *User Interface Management System* are used. The former is used to create the computational part and manages the application at run-time, while the latter is used to create the graphical user-interface (UI) part, manages the user-interface at run-time and is used for "connecting" the two parts.

4.2.1 Semantics-directed UI construction

Since we are concentrating on the user-interface component of an interactive programming environment, we will assume a set of functions F_i to be present in a computational component. Basically, when designing a UIMS, one has to consider the following questions:

(1) What is the graphical appearance of the UI and how should it behave when the user interacts with it?

(2) Which functions F_i should be available to the user via the UI?

(3) When is F_i available to the user, i.e., what are the enabling conditions for F_i?

(4) When F_i is available, what is the source of its arguments and what is done with its result?

In our approach, the first question (what is the "look and feel" of the UI) is answered *implicitly*, since we will only use *logical UI objects* such as buttons, menu entries, and the like. The "look and feel" of these objects is *predefined* and therefore contributes to our primary goal of achieving uniformity.

For all interactions with the user, we use instances of GSE as described in previous chapters. In each instance, all language-dependent parts are parameterized by means of a syntax definition. Furthermore, each instance has a default graphical user-interface. Thus here too the "look and feel" is predefined. Buttons, menus, etc. can be added to the UI of one or more editor instances. These UI extensions are placed at a *predefined* position in the editor's window. As a result, a *dialogue developer* (a human being using a UIMS), can concentrate on the *semantic* aspects (which functions are available, what are their enabling conditions, how are their arguments and results handled) and is freed from all *syntactic* aspects (handling mouse or keyboard input, defining graphics, etc.).

We can answer questions (2) and (3) by distinguishing two types of functions in the computational component:

- state inspection functions; and
- state manipulation functions.

The *state* of an application is defined as the current data stored in the application. *State inspection functions* inspect the current state, while *state manipulation functions* change and, in some cases, also inspect it. The former type is used to define the enabling condition of a function associated with a UI object, while the latter type defines the associated function itself. The distinction between state inspection functions and state manipulation functions was made earlier by D.L. Parnas, who called them V-functions (Value delivering functions) and O-functions (Operate functions) respectively [Par72]. V and O functions were first used in the context of user-interfaces by R. Eckert [Eck80].

Consider, for example, a word processor with "cut", "copy" and "paste" functions, each made available to the user via entries in a menu. The "paste" function inserts text stored in a buffer. When there is no text in the buffer, the "paste" function is not available. Thus the "paste" menu entry must be changed, with respect to functionality (it must not respond at all), as well as graphical appearance (to provide semantic feedback to the user: "paste" is currently not enabled). The point here is that both the functionality and the graphical appearance depend on data stored in the application. A function implementing the "is there any text in the buffer?" question is thus a typical state inspection function. The "paste" function itself is a typical state manipulation function, since it changes data stored in the application.

Finally, question (4) is answered by stating that an *argument* of a function is either a part of the structure maintained by an editor instance, or the result of a function call. The *result* of a function call is either used as an argument of another function or assigned to a part of the structure maintained by an editor.

Summarizing, a UIMS controlling the UI at run-time must be able to inspect the application's state and call state manipulation functions. Conversely, the application has to notify the UIMS whenever its state changes, since in that case the UIMS must re-consider the availability of functions. These observations form the basis of our approach.

A UI is described as a list of logical UI objects. Each object description is a list of *condition-action* pairs, where a condition is a state inspection function and an action is a *list* of state manipulation functions. Note that a list of such functions constitutes a new state manipulation function. To avoid confusion, we say that an action consists of *statements*. By allowing more than one condition-action pair, we obtain a situation where the same UI object can be associated with different actions, depending on the state of the application.

4.3 Motivating examples

We present a number of motivating examples, each illustrating a typical problem which a UIMS has to solve.

4.3.1 Enabling of a function depends on application state

We already discussed the menu containing an entry "paste" that should only be enabled when the "cut-copy-paste" buffer is non-empty. Related examples are the "copy" and "cut" functions that should only be enabled when the user has selected a piece of text.

Another example is a function that performs a program transformation. In many cases, these transformations are based on a conditional rule which states that a program part matching the left-hand side of the rule can be replaced by the right-hand side of the rule or vice versa. Here again, a transformation function should be disabled when its condition is false.

4.3.2 Function needs user input

Consider a UI for a simple database for names, addresses and telephone numbers, and a UI containing —among others— a "lookup" button. When this button is pushed, the user is first asked to indicate whether a telephone number or an address has to be looked up. Next, the user is asked to select a person's name. Finally, the appropriate function in the computational component is called and its result is presented to the user.

This example illustrates two different kinds of user input. In the first dialogue, the user is asked to select an item from a *fixed* set of values (lookup of a telephone number versus lookup of an address), while in the second one, the user is asked to select a name from a *variable-sized* set of values which can only be computed by the application at run-time.

4.3.3 Function with several input sources

In the previous example, the list of names from which the user picks an item, can be computed from a single source: the database file. There are also situations where multiple sources are used in a computation at run-time, for instance, a compiler which handles multiple source files. In this case, the system first has to collect all necessary input values and then pass them to the "compile" function in the computational component.

4.3.4 Implicit invocation

All examples presented so far are instances of what we call *passive* tools which are only activated upon explicit user request. There are also *active* tools which are invoked automatically whenever appropriate. An example of such an active tool is a command line interpreter, which automatically executes a command when the user has completely entered it. Other examples are a typechecker which is automatically called after *each* modification of a program and an automatic source file utility that maintains a list of changed files for the benefit of efficient recompilation.

4.3.5 Repeated invocation

The above source file utility will do a fixed number of calls to a compiler: once for each modified source file.

Other computations may be incorporated in an endless loop that is started and stopped upon explicit request by the user. This situation is found in, for instance, automatic demonstrations of systems. A demonstration typically consists of a number of operations repeated over and over again, until it is ended by the user. So, a user starts the demonstration, watches what happens for some time and then stops it. Another example is a test program using randomly generated input values. The test program runs a given program with an input value, shows the result, generates another input value, runs the program again, and so on.

4.4 Abstract representation of application domain

At an abstract level, one can describe an interactive programming environment as a collection of —possibly cooperating— syntax-directed editors. In each editor resides a well-typed abstract syntax tree (also called "term") and the computational component offers a set of semantic operations on these terms. Note that all these editors are generic and can be used to edit programs in different languages.

Each editor has a *focus* designating a subtree of the tree residing in it. Editing commands, such as replacing a placeholder by a template, are all relative to the focus subtree. The focus may be moved to another subtree by invoking a *navigation* command, such as "go to the next child".

We represent a UI as a set of logical UI objects which are added to the default UI of one or more editors. Each UI object consists of a *name*, a *type* (button, menu entry, etc.) and a set of *rules*. A rule is a "condition-action" pair, where the condition is a state inspection function and the action is a state manipulation function. When UI objects are added to an editor, we say that it *uses* these rules.

Our application domain can be described as a set of editors, each described by a quadruple of the form (*Name*, *Rules*, *Tree*, *Subtree*). The *Name* and *Rules* are *static*, since they are defined "at editor creation time". The *Tree* and *Subtree* are *dynamic* since the user manipulates them interactively by editing, by navigating, or by applying a semantic operation.

4.4.1 Abstract representation of state

A syntax-directed editor E_i is characterized by $\langle N_i, R_i, T_i, S_i \rangle$, where N_i is the name of editor E_i, R_i is a set of rules used by E_i, T_i is the tree residing in E_i and S_i is the focus subtree of E_i. The *state* of the programming environment is then defined as a set $E = \bigcup E_i$ of syntax-directed editors.

This definition immediately suggests what state inspection functions and state manipulation functions are needed. State inspection functions are:

- Does an editor with name N exist?
- Is S_i of type X?
- Is T_i of type X?
- Does S_i match with tree pattern P?
- Does T_i match with tree pattern P?

These functions are all of type Boolean, and Boolean operators can be used to form new state inspection functions.

State manipulation functions are:

- Add editor E_k to E;
- Delete editor E_k from E;
- Change T_i to T'; and
- Change S_i to S'.

State manipulation functions thus change the number of editors or change a (sub)tree in an editor. Any combination of these functions also constitutes a state manipulation function.

Typical examples of (sub)tree changes in our application domain are:

- Invoking a navigation command in editor E_i.
- Editing T_i or S_i.
- Assigning the result of a function F_i to T_j or to S_k.

4.4.2 A user session

When and how can the user activate state manipulation functions? Let the system be in state E, and let R be the union of all rules R_i in E. Let r be an element of R, where r is a list of condition-action pairs denoted by $\{[C_1, A_1],\ [C_2, A_2],\ .....,\ [C_n, A_n]\}$. The set of rules available to the user via the UI, when the application is in state E, R_E, is defined as:

$R_E = \{r \in R \mid \text{some } C_i \text{ in } r \text{ is true}\}$.

Let *void* denote the empty action and let the function *action* be defined as:

action(r, E) $= A_i$ if C_i is true and $\forall_{j<i}\ C_j$ is false,
action(r, E) $=$ *void* otherwise.

We are now able to describe a user session. Let the application be in state E_0. Then, the user selects a rule r from R_{E_0} and action(r, E_0) is applied, leaving the application in state E_1. Next, the user selects a rule r from R_{E_1}, action(r, E_1) is applied, bringing the application in state E_2, etc. Note that after a state transition form E_i to E_{i+1}, the set $R_{E_{i+1}}$ has to be computed.

4.5 SEAL: an experiment in UI definition

In this section we introduce SEAL (Semantics-directed Environment Adaptation Language) a dedicated language for connecting the user-interface of an editor with functions in the computational component. A UI description in SEAL, or *script*, thus "seals together" the two components of an interactive system. The computational component is defined in the ASF+SDF formalism, which we describe in the next section. In the section thereafter, we introduce SEAL's predefined functions. Then, we will show how the examples of Section 4.3 can be defined in SEAL. Finally we will briefly describe how SEAL is implemented.

4.5.1 The ASF+SDF specification formalism

ASF+SDF [HK89] is a modular algebraic specification formalism developed in the GIPE project. It is the result of merging SDF (Syntax Definition Formalism) [HHKR89, Rek92] used to define the syntax of a language, and ASF (Algebraic Specification Formalism) [BHK89] used to define its semantics.

A simple ASF+SDF specification for the language of Boolean expressions consists of two modules, one defining the syntax of the language (Figure 4.1) and the other defining its semantics[2] (Figure 4.2).

Module `Boolean-syntax` (Figure 4.1) defines the syntax by introducing a *sort* `BOOL`, which contains two *constants* `true` and `false`. Furthermore, the functions `and`, `or`, and `not` are defined as well as parentheses. The attribute `left` declares `and` and `or` as left-associative functions and the *priorities* define function grouping when no parentheses are present. Module `Boolean-syntax` imports the module `Layout`, in which layout characters —spaces, tabs, newlines, etc.— as well as a comment convention are defined. We omit this module.

Module `Boolean-semantics` (Figure 4.2) defines the semantics of the Boolean language by introducing a *variable* of sort `BOOL` and *equations* which define equalities on Boolean *terms*. Variables are introduced by the naming scheme `b[0-9]* -> BOOL` which states that variable-names starting with the character `b` followed by a sequence of zero or more digits will be used to denote variables of sort `BOOL`. Although not used in this example, an ASF+SDF equation may have a *condition* expressing that the equation

[2]This strict separation between syntax and semantics is not required by ASF+SDF, but we will use it in examples later on (Section 4.5.3) where we define transformations on Boolean expressions.

```
module Boolean-syntax
imports Layout
exports
  sorts BOOL
  context-free syntax
    true              -> BOOL
    false             -> BOOL
    not BOOL          -> BOOL
    BOOL and BOOL     -> BOOL  {left}
    BOOL or  BOOL     -> BOOL  {left}
    "(" BOOL ")"      -> BOOL  {bracket}
  priorities
    not > and > or
```

Figure 4.1: Module Boolean-syntax.

```
module Boolean-semantics
imports Boolean-syntax
  variables
    b[0-9]*           -> BOOL
equations
  [1] true  and b = b
  [2] false and b = false
  [3] true  or b  = true
  [4] false or b  = b
  [5] not true    = false
  [6] not false   = true
```

Figure 4.2: Module Boolean-semantics.

is only applicable when the condition holds. The general form of a condition is a *syntactic equality* (or *inequality*) on terms. Note that the syntax defined in `Boolean-syntax` is *used* in the equations of `Boolean-semantics`.

4.5.2 Overview of SEAL

In this section, we present the basic notions and functions of the SEAL language. Its complete syntax in SDF may be found in Appendix A.

Focus expressions

There are three basic notions in SEAL: *focus-expressions*, *variables* and *editor-names*. A focus-expression determines a source or destination subtree of an editor. Its simplest form, the keyword `focus`, denotes the focus of the editor instance to which the UI object is added. The general format:

```
<editor-name>.focus <moves>
```

denotes the focus of the instance with name `<editor-name>`, but first moved to the position indicated by `<moves>`, a list of elementary tree moves, such as `up`, `down`, `next`, `previous` and `root`. Editor names are equal to the corresponding filenames.

Predefined functions in conditions

In conditions, the following functions may be used:

- pattern matching, as in `<focus-expression> matches <pattern>`;
- meta-variable checking, as in `<focus-expression> is-a-metavar`;
- sort checking, as in `<focus-expression> is <sort-name>`;
- calling a function, as in `<module-name> : <function-call>`; and
- any of the above, combined with Boolean operators `and`, `or`, or `not`.

A *pattern* is a string containing *meta-variables*, which are textual representations of placeholders. For instance, a pattern matching a focus positioned at an `or` construct in the Boolean language, is denoted by: `"<BOOL> or <BOOL>"`. Note that a meta-variable is denoted by a sort-name between the characters "<" and ">".

Meta-variables are also used as placeholders and may also appear in texts residing in an editor instance. A "meta-variable check" is therefore incorporated to recognize a focus positioned at such a placeholder.

Finally, a function in the computational component may be called. The `<module-name>` part of the function call notation is used to denote the *context* in which the function must be evaluated. This is due to the modular set up of the ASF+SDF formalism: the same term may have different semantics when evaluated in the context of different modules. For example, the term `true or false` evaluated in the context of the module `Boolean-syntax` has *itself* as semantics, because there are no equations in that module expressing otherwise. Evaluating the same term in the context of the module `Boolean-semantics`, however, leads to the semantics `true` since this is defined by the equations in that module.

Predefined functions in actions

In actions, the following state manipulation functions, or *statements* in SEAL terminology, may be used:

- moving a focus;
- assigning a term to a variable or a destination determined by a focus-expression;
- computing a term by calling a function in the computational component;
- creating an editor instance to be used as text-input or term-output window;
- selecting a (sub)term of a certain sort in an editor instance; and
- (conditional) looping over any number of statements.

Allowed focus moves are the same as in conditions, plus the statements `save` and `restore` for saving the focus position and restoring it to a saved position in the tree. This provides a means to make temporary excursions with the focus during the execution of actions.

Arguments of functions may be obtained using focus-expressions or variables. Function results may be assigned to focus-expressions or variables.

New editor instances can be used as text-input or term-output window. The text-input statement is convenient for asking the user to select an item from a fixed set of values that is known a priori. It is normally used in combination with a `select` statement (described below). Its format is:

```
create(<editor-name>, <module>, <text>, <sort>)
```

where `<editor-name>` is an editor name, `<module>` is the name of the module defining the syntax of this editor, `<text>` is a list of strings forming the text to be stored in the editor's text-buffer and `<sort>` is the name of the sort of the root of the editor's tree. At run-time, a text-input statement creates an editor with name `<editor-name>`, with text `<text>`. The text is parsed using the syntax defined in `<module>` and the parser is requested to use `sort` as the sort of the root.

The term-output statement is normally used to present the result of a function to the user. Its format is:

```
create(<editor-name>, <term>)
```

where `<editor-name>` is an editor name and `<term>` is any term. At run-time, a term-output statement creates an editor with name `<editor-name>`, containing tree `<term>`. The tree is pretty-printed thus forming the text. The syntax definition to be used by a term-output instance is *derived* from the term. If the term is a focus-expression $N.focus$, the syntax definition of editor N is used. If the term is the result of a semantic computation `Module : functioncall(...)`, the syntax defined by $Module$ is used. In text-input instances as well as in term-output instances, we also allow a *term* to be used as the editor's name. This provides a means to *compute* a name of a text-input or term-output instance using semantic computations on names.

The user may be asked to point at a subtree of certain sort in an editor instance, in which case the `select` statement must be used. Its format is:

```
select(<editor-name>, <sort>)
```

where `<editor-name>` is an editor name and `<sort>` is a sort. At run-time, the `select` statement pops up the window of the editor `<editor-name>`, and a dialog is displayed asking the user to point at a subtree of sort `<sort>` using the mouse. Upon clicking the mouse, the smallest subtree S is calculated such that the text corresponding to S contains the character pointed at by the user and S is of sort `<sort>`. If such an S can not be calculated, the `select` statement reports an error and asks the user to point again. Finally, `select` returns the subtree chosen. Instead of an `<editor-name>` one can also use a term (as with text-input and term-output instances) as a means to use computed editor names.

Finally, we allow conditional looping over any number of statements. For the condition, we allow all constructs as found in SEAL conditions as well as *moves*. Moves are used when a function needs the current focus position as an argument. At first sight, this may seem strange, but a computation may need the position of the focus as an argument. Consider, for example, a function computing the body of a Pascal procedure P, when the focus is located at a call to P. Then, due to Pascal's scoping rules, there might be more than one procedure P in the whole program, so the result of the function depends on the current focus position. Thus, a move in a loop condition is a *test* whether or not this move is possible.

Predefined UI objects

We distinguish five types of UI objects: *buttons*, *start-stop buttons*, *menu entries*, *menus*, and *active tools*. Note that this implies that all other types of UI objects, such as dialog-boxes, radio buttons, file browsers, etc., must be simulated by using editor instances. This approach was earlier suggested by others, see, e.g., [DS90]. We will now present the general format of each type of UI object in SEAL.

Buttons are defined by:

```
button <name> when <cond> enable <action>
```

This defines a button with label `<name>`. The `when <cond>` part is optional. A special kind of button is the *start-stop* button. When pushed, it repeats the action as long as `<cond>` holds. Repetition can be stopped by the user by pushing the button again. Between the statements of an actions, repetition may be interrupted as well. Start-stop buttons are denoted by prefixing the keyword `button` with the keyword `start-stop`.

Menu entries are defined by:

```
menu entry <name> in <menu> when <cond> enable <action>
```

This defines an entry with label `<name>` as part of a pulldown menu with label `<menu>`. The `when <cond>` part is optional. For convenience, entries in the same menu may be grouped using:

```
menu <menu> :
<name-1> when <cond-1> enable <action-1>
...
<name-N> when <cond-N> enable <action-N>
```

Finally, an active tool with name `<name>` is defined as:

```
active tool <name> when <cond> do <actions>
```

The `when <cond>` part is optional.

4.5.3 Examples in SEAL

Now we are in the position to define some of the examples discussed earlier in Section 4.3.

Rule based transformations

Consider the Boolean language presented in Section 4.5.1 and let us add a menu `Transformations` with an entry to which a function is connected implementing De Morgan's laws. The SEAL script for such an entry is:

```
menu entry DeMorgan in Transformations
when
  Transform : de-morgan-possible(focus)
enable
  focus := Transform : de-morgan(focus)
```

This menu entry requires an additional module called `Transform` which is defined as:

```
module Transform
imports Boolean-syntax
exports
  context-free syntax
    de-morgan-possible(BOOL) -> BOOL
    de-morgan(BOOL)          -> BOOL
equations
  [1] de-morgan-possible(not(b1 or b2)) = true
  [2] de-morgan-possible(not b1 and not b2) = true
  [3] de-morgan(not(b1 or b2)) = not b1 and not b2
  [4] de-morgan(not b1 and not b2) = not(b1 or b2)
```

Note that module `Transform` imports the module `Boolean-syntax`, not `Boolean-semantics` since we are defining a syntactic transformation only. Now, whenever the focus of an editor instance using `Boolean-syntax` as syntax definition is positioned such that evaluating `de-morgan-possible (focus)` in the context of the module `Transform` yields `true` the `DeMorgan` menu entry is enabled by adding the entry to the menu `Transformations`. Of course, it is not added when it is already there and furthermore, menus without any entries are deleted form the UI so adding an entry might imply adding a menu to the UI. Selecting `DeMorgan` replaces the focus by the result of evaluating `de-morgan(focus)`, thus by the right-hand side of either equation [3] or [4].

Database lookup

The SEAL script for the simple database lookup example[3] presented in Section 4.3.2 is:

[3]Disclaimer: this example has as only purpose to illustrate several features of SEAL. With a similar effort, however, we could define a *usable* interface for this example

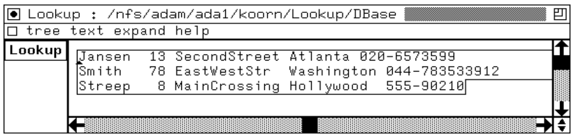


Figure 4.3: The editor containing the database with added `Lookup` button.

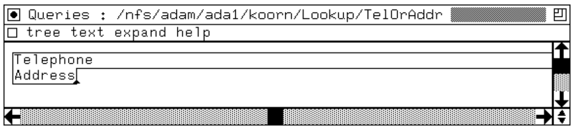


Figure 4.4: The editor displaying QueryTypes, before selection.

```
button Lookup
enable
   create("TelOrAddr", Queries, "Telephone" "Address", QueryTypes);
   TheQueryType := select("TelOrAddr", QueryType);
   AllNames := Queries : get-names("DBase".focus root);
   create("Names", AllNames);
   TheName := select("Names", Name);
   Result := Queries : lookup("DBase".focus root, TheQueryType, TheName);
   create("LookupResult", Result)
```

Pushing `Lookup` (Figure 4.3) results in first creating a new editor instance `TelOrAddr`, which uses the syntax defined in module `Queries` and contains the two strings `Telephone` and `Address` as text (Figure 4.4). When this text is parsed, it forms a tree of sort `QueryTypes`. Then the user is asked to point at a subtree of sort `QueryType` in the just created instance, by the `select` statement (Figures 4.5 and 4.6). The selected subtree (the tree representation of either `Telephone` or `Address`) is stored in the variable `TheQueryType`. Next, all names in the database are computed and shown to the user in a new instance `Names` (Figure 4.7), from which the user selects a subtree of sort `Name` (Figures 4.8 and 4.9). Note that this computation is implemented by calling the function `get-names` in the computational component. Finally, the result of looking up either a telephone number or address is presented in the editor instance `LookupResult` (Figure 4.10).

Compilation of multiple source files

Consider the compiler presented in Section 4.3.3. Let the UI of this compiler use an editor in which the user can write sentences like `compile a.x b.x`

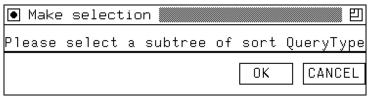


Figure 4.5: A dialog box asking to select a `QueryType`.

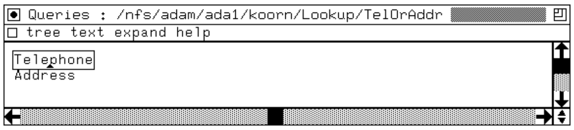


Figure 4.6: The editor displaying QueryTypes, after selecting `Telephone`.

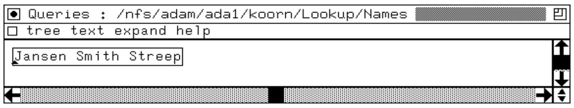


Figure 4.7: The editor displaying all names in the database, before selection.

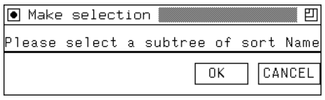


Figure 4.8: A dialog box asking to select a `Name`.

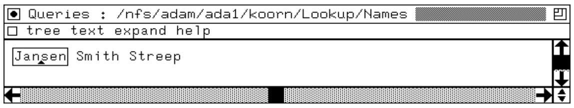


Figure 4.9: The editor displaying all names in the database, after selection.

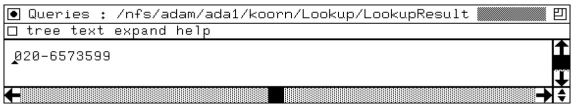


Figure 4.10: The editor displaying the telephone number of Jansen.

`c.x`, where `a.x`, `b.x` and `c.x` are names of source files. We add a button to the UI of the editor using:

```
button Compile
enable
   Filenames := Interface : get-filenames(root);
   All := Interface : empty-source();
   while Interface : files-left(Filenames) do
      NextName := Interface : first-file(Filenames);
      Filenames := Interface : next-files(Filenames);
      All := Interface : concat(All, NextName.root)
   od;
   Code := Compiler : compile(All);
   create("CodeView", Code)
```

Pushing the `Compile` button results in deriving the list of filenames, concatenating all files, compiling them and showing the generated code in a separate window.

Command line interpreter and automatic typechecker

Let the command line interpreter presented in Section 4.3.4 have two commands: `compile filename` and `typecheck filename`. The SEAL script for this active tool is:

```
active-tool Interpreter
when Interface : is-typecheck-command(focus root)
do
   FileName := Interface : get-filename(focus root);
   Errors := Typechecker : tc(FileName.focus root);
   create("ErrorView", Errors)
when Interface : is-compile-command(focus root)
do
   FileName := Interface : get-filename(focus root);
   Code := Compiler : compile(FileName.focus root);
   create("CodeView", Code)
```

The automatic typechecker presented in Section 4.3.4 can be defined as:

```
active-tool Typechecker
do
   Errors := Typecheck : tc(focus root);
   create("ErrorView", Errors);
```

Test program

The test program, described in Section 4.3.5, is our final example. It uses a start-stop button that repeats the action as long as its condition holds. Repetition may be stopped by pushing the button again. Between the statements of an action, repetition may be interrupted as well. The SEAL script for a `Test` button is:

```
start-stop button Test
enable
   Input := Tester : generate-first-value();
   Program := focus root;
   while do
      Result := Evaluator : run-program(Program, Input);
      InputAndResult := Tester: make-pair(Input, Result);
      create("ResultView", InputAndResult);
      Input := Tester: generate-next-value(Input);
   od;
```

4.5.4 Miscellaneous issues

We did not yet describe which logical UI objects are added to what editors. SEAL scripts are tagged with the name of an ASF+SDF module. All UI objects described by a SEAL script tagged with name N, are added to the UI of editors parameterized with the syntax defined by N. For example, the `DeMorgan` menu entry presented in Section 4.5.3 is part of a script for the Boolean language, defined by the module `Boolean-syntax`. The UI objects for this language are defined as:

```
Configuration for language Boolean-syntax is

menu entry DeMorgan in Transformations
   ...
button ...
   ...
```

4.5.5 Implementation

To conclude our presentation of the SEAL language, we describe its implementation. We discuss how scripts are developed, how SEAL's typechecker, its compiler, and its run-time system are implemented, and how the system operates.

SEAL environment

SEAL scripts can be typechecked and compiled using a programming environment which is written in the ASF+SDF formalism. In this environment, editors using the SEAL language have two additional buttons for typechecking and compiling which were created using SEAL.

Until recently, the generated environment for SEAL could not be used as a separate, stand-alone tool. Changing a script thus required re-generating the environment for SEAL, taking up to half an hour. Furthermore, this set-up required quitting and restarting the computational component in many cases as well due to excessive memory requirements. This was resolved by compiling the ASF+SDF specifications of the SEAL typechecker and compiler to C source code using techniques described in [KW93] and [Bra93]. Creating or changing a SEAL script without quitting the computational component is now done in less than half a minute *including* the removal of outdated user-interface objects and the addition of new ones. As a result, users can adapt the user-interface of their environments easily and may experiment freely. Adding the ASF+SDF module defining only the SEAL syntax to their specification is all that is required.

SEAL typechecker and compiler

The SEAL typechecker checks static constraints and, if any, reports them in a separate window. These constraints include, for instance, using variables that are not yet defined and restoring a focus position which was not saved before.

The result of a compilation is a LeLisp [LeL91] source file in which functions are defined implementing conditions and actions. The body of these functions consist of calls to interface functions defined by the run-time system which is described below. Furthermore, a compilation result contains a *configuration function* for the language for which the script was written. Calling this function leads to the addition of an entry to the so called *SEAL table* which is stored in the data structure of the ASF+SDF Meta-environment. The entry contains the list of logical UI objects in internal format. This format contains the name and the type (button, active-tool, etc.) of the object as well as the names of the functions implementing conditions and actions.

Part	total	SDF	ASF	LL
Run-time and initialization	3688	-	-	3688
Typechecker	3939	954	2985	-
Compiler	786	1007	2993	-
TOTAL	11627	1961	5978	3688

Table 4.1: Overview of sizes of source code involved in SEAL.

SEAL run-time system

To minimize the amount of code generated by the SEAL compiler, we have defined a "run-time" package implementing interface functions for sort checking, focus movements, term-reductions, etc. Furthermore, there is a small "mapping language" in which a user describes from where (i.e., from which directories) modules should be read by the ASF+SDF Meta-environment, where that system should look up compiled SEAL scripts and also what filename-extension is used by editors of a certain language. Texts written in the mapping language, are saved in a special file called `.seal` in the user's home directory.

When creating an editor with name $N.x$, the system inspects the `.seal` file. If it finds an entry where the filename-extension equals `x`, it loads all necessary modules, it loads the necessary compiled SEAL scripts, and calls the configuration function for the language involved. Finally, the editor instance is created with the specified UI objects added to its UI.

Whenever the environment's state is changed, for instance when a focus is moved, the editor where the change occurred reports this to the ASF+SDF Meta-environment. The SEAL run-time system is then given control. It inspects the SEAL table, evaluates conditions, and finally updates the *names* of the functions implementing actions if necessary. A later invocation of the UI object by the user then leads to calling the appropriate function. However, when the UI object is an active tool, the appropriate function is called immediately.

Quantification of code involved

How much code was involved to implement SEAL? To answer this question we use the same counting scheme as before (cf. Section 3.7). The sizes of the sources involved in SEAL are listed in Table 4.1. The implementation of SEAL can be subdivided into three parts: typechecker, compiler, and run-time code. Some of these parts share code and a considerable amount

Part	total	SDF	ASF	LL
Run-time management	2902	-	-	2902
Initialization	786	-	-	786
TOTAL	3688	-	-	3688

Table 4.2: Overview of sizes of SEAL source code involved at run-time.

Part	total	SDF	ASF	LL
Shared (with compiler)	1708	485	1223	-
Generated	634	49	585	-
Rest	1597	420	1177	-
TOTAL	3939	954	2985	-

Table 4.3: Overview of sizes of source code involved at typechecking-time. Generated code was produced by the generator described in [Bra93].

of code was generated. The sizes of the sources involved in each part are therefore listed in Table 4.2 through 4.4.

Part	total	SDF	ASF	LL
Shared (with typechecker)	1708	485	1223	-
Generated	127	30	97	-
Rest	2165	492	1673	-
TOTAL	4000	1007	2993	-

Table 4.4: Overview of sizes of source code involved at compile-time. Generated code was produced by the generator described in [Bra93].

4.6 Related work

We briefly discuss representational schemes found in UIMSs as well as the mechanisms used to connect the user-interface to the application. For an extensive survey of UIMSs we refer to [HH89, Hee92].

4.6.1 Representational schemes

We briefly discuss representational schemes found in UIMSs and describe how a dialogue developer creates a description.

The *state*[4] *transition diagram approach* [Jac86] is mainly used for coding so called *sequential dialogues*. Given a start node, the UI asks the user for input. After validating the input, a function in the computational component might be called and the UI changes its state. Most systems using this technique present the dialogue developer with a graphical editor to edit the diagrams, using stepwise refinement for sub-dialogues. Others use a textual description of states and transitions.

Basically, systems using the *grammar approach* [Mor81, Bos88, SY88] for coding dialogue are similar to systems using state transition diagrams. The main difference is that the former always use a textual description, while the latter use a graphical description. Actions and input validations are interleaved with terminals or non-terminals of the grammar.

User-interfaces built with systems using an *abstract event approach* [Hil86, JMB+93] are based on a window environment in which the user is able to select the next step in the dialogue sequence, for instance by pushing a button using a mouse device. Thus, the user does not respond to a question of which the answer determines the next step. This is called a *non-sequential* dialogue. Systems using an event description approach are more suitable for coding this type of dialogue than those using state transitions or grammars. The basic idea can be described as: "when X happens, do Y", where X is an event and Y is the corresponding action. Note that, in principle, this approach allows concurrent evaluation of actions.

In the *direct manipulation interface approach* [WR82, SM88, Mye90, MGD+90, MSK90, BL90, Rem92] the dialogue developer uses a drawing package to build a UI. Typically, the drawing package has a direct manipulation [Shn83] user-interface. In some systems using this approach, the dialogue developer may connect a function to a user-interface object by selecting one from a list of all functions present in the computational component. In other systems in this category, result values of these functions (and also internal functions, such as activation of a button) can be used as arguments of other functions. The dialogue developer then uses icons representing functions and drags the "output" of one function icon to one of the "inputs" of another function icon.

Despite the fact that we covered most approaches to representing UIs, there are still a few worth mentioning. For instance CLG [Mor81] can be used to describe UIs, but CLG covers far more than the description only. Furthermore, there are "toolkits" extended with a UI definition language in which the layout of windows is textually described.

[4]This notion of *state* should not be confused with states as defined in Section 4.2.1

4.6.2 Connection mechanisms

We describe some mechanisms used by UIMSs to connect the user-interface component (UI) to the computational component (CC). We concentrate on the question: how well are the components separated from each other?

The use of *active variables* is found in Peridot [Mye90] and in its successor Garnet [MGD+90]. These are special purpose data structures that may be changed by either the UI or the CC. If the value of an active variable is changed by the UI, the CC is notified and vice versa. This provides separation of the both components involved, since the response of component A to changing the variable's value by component B is hidden from B. When such variables are used for the run-time enabling or disabling UI objects, an extension of the UI requires an extension of the CC: in some cases the CC must change a variable's value to ensure the enabling (disabling) of the newly added UI objects. In other words, the CC "knows" which objects are present in the UI.

Mapped variables [WL] are a variant of active variables. An extra level of indirection has been added to improve flexibility. The indirection consists of applying an arbitrary function to the value of an active variable before the other component is informed. In [WL], these mapped values are written in a database. Both components poll the database frequently to inspect if any changes have been made. Mapped variables suffer from the same disadvantage as active variables: the CC "knows" which objects are present in the UI.

Pre- and postconditions are suggested in [GF92]. This paper appeared at a time where SEAL was being implemented and the technique suggested shows some remarkable resemblances. Pre- and postconditions are associated to each UI object. Preconditions are used to enable or disable the object at run-time and are thus similar to SEAL's conditions. The postconditions are used to manipulate *predicates*, which are written on a "blackboard" after the execution of the action of the object. These predicates are used in the preconditions. Note the similarity between the "blackboard" used here and the database approach of [WL]. The separation of the UI and the CC is, in our view, somewhat obscured by allowing the CC to manipulate the blackboard. The use of postconditions to manipulate the blackboard's contents has another disadvantage: addition of a new object to the UI might require an adaptation of the postconditions of existing objects.

Taps [Ber92] are yet another way to connect UI and CC. In this scheme, every command invoked by the user (callback) is intercepted and may trigger one or more taps which manipulate the UI. Here too, the separation of UI and CC is somewhat obscured by allowing the CC to trigger taps as well.

Abstract events [Hil86, JMB+93] provide a way to separate UI and CC such that the one does not "know" the contents of the other. In this scheme, any change leads to broadcasting a message (event) reporting the change. Each component in the interactive system may then respond to the message or it may simply ignore it. Note that the sender of the message, i.e., either the CC or a UI object, does not "know" who is listening. The separation is not 100% complete however, since adding a new object to the UI requires an adaptation of the network transporting the message: the CC must now listen to this object as well, or must inform the object when it should enable (disable) itself. The latter may require adapting existing code in the CC.

4.7 Discussion and conclusions

4.7.1 Summary

We have argued here that a UIMS should incorporate availability of functions, and we have shown that this can be achieved by dividing the set of functions that the computational component offers into two sets: state inspections and state manipulations. The notion of state, i.e., the current data stored in the application, is the key to this division.

To each UI object (button, menu entry, etc.) managed by the UIMS at run-time, we associate a condition (state inspection) and an action (state manipulation). The action is made available to the user, provided that its corresponding condition holds. As a consequence, the application must notify the UIMS whenever its state changes, since the UIMS must then re-consider the availability of actions using the corresponding conditions.

In the application domain of interactive programming environments based on a collection of syntax-directed editors, a UI can be described as a list of logical UI objects which are added to the default UI of an editor. As an experiment, a language to define UIs used in the ASF+SDF Meta-environment was developed (SEAL). All necessary run-time code is generated from the UI description by the SEAL compiler. Logical UI objects offered by the language include buttons, start-stop buttons, menu entries, menus, and active-tools. As a result, all objects, including editors, have a predefined "look and feel". SEAL shows the feasibility of our approach in the above mentioned application domain.

4.7.2 Advantages

There are four advantages to our approach.

First, a dialogue developer is freed from all syntactical aspects such as handling mouse or keyboard input, and defining graphics. As a consequence, a dialogue developer can concentrate on the semantic aspects (which functions are available in the UI, what are their enabling conditions, how are their arguments and results handled).

Second, a complete separation of the computational component and the user-interface component of an interactive system is achieved, even for the aspect of availability of functions. This is due to the fact that the application knows only that there is a UI, some objects of which may depend on its state. However, the application does not know *which* objects are involved or *how* they depend on its state. It merely reports a state change to the UIMS, after which it is the responsibility of the UIMS to take appropriate action. Code managing UI objects at run-time is thus completely separate from application code. Furthermore, new UI objects may be added freely to the UI description. That is, such an addition never requires an adaptation of the description of existing objects, as may be the case when postconditions are used.

Third, our approach yields UIs with a uniform "look and feel". As a result, once a user is familiar with such a UI, a new system can be learned very quickly. On the other hand, this uniformity may also be considered a disadvantage because a user is unable to customize the "look and feel" of the UI. We consider uniformity of UIs more important.

Finally, a dialogue developer does not have to write *any* code since all code is generated from the UI description. Even the code "connecting" the computational component and the user-interface component is either generated or is part of the default UIMS code.

4.7.3 Disadvantages

As we have seen, the uniform "look and feel" of UIs may be considered a disadvantage. Furthermore, there are two obvious and two somewhat more technical disadvantages.

Two obvious disadvantages are the fixed, and thus limited, set of UI objects and the restriction to a particular application domain.

A more technical disadvantage is the inability to generate a UI based on direct manipulation [Ols87, pg. 97-101]. One of the basic aspects of a direct manipulation UI is providing semantic feedback (by changing the graphical appearance) *during* the manipulation of an object by the user. Our approach only provides semantic feedback *after* the user has manipulated an object (i.e., changed the state). To be able to generate direct manipulation UIs, the notion of state should probably be extended, or such a notion should be made part of the UI. This is clearly an area for future research.

Another technical disadvantage is the obligation to generate sequential code. To illustrate this, let us assume we generate concurrent code and consider, for example, two buttons which are enabled. When a user activates both buttons in a short time interval, both try to change the application's state; one of them may fail to do so because the other button has already changed the state. Thus after the evaluation of the condition of one of the buttons, the state is changed by the other button, but since the action has already been invoked, the re-evaluation of the condition comes too late, possibly causing the action to fail. In the case of generating sequential code, the action of the button first-started is performed, after which the condition of the second button is automatically re-considered leading to its disablement. Thus in general, generating concurrent code for more than one object requires in any case that its condition is independent of the action of any other object. The independence check needed can probably be derived from the UI description, but this too will require more research.

4.7.4 Final remarks

SEAL is our first experiment in UI definition and UI generation, and we only presented some simple examples of its use. We did not discuss its practical merits when defining "real-life" environments, but we will do so in the next chapter. However, we have already shown that the complete separation of the computational component and the user-interface component is a major advantage, as is the absence of any hand-written code. The users of the ASF+SDF Meta-environment are not aware of the disadvantages as they view SEAL as an extension rather than as a UIMS with restrictions.

Returning to our starting point, the question of how to connect semantic tools to a syntax-directed user-interface, we could say that such tools can be connected by using code generated from a textual description, in combination with an extensible syntax-directed user-interface and a UIMS which controls the interface at run-time.

Chapter 5

Generating applications with SEAL: some case studies

We present five case studies of user-interface generation with SEAL. Each illustrates a typical application area of the generation of interactive programming environments by the ASF+SDF Meta-environment. The practical value of the SEAL formalism is assessed and a number of suggestions are made to improve or extend it.

5.1 Introduction

In the previous chapter we introduced SEAL, a dedicated user-interface definition language for the ASF+SDF Meta-environment. The main topic of this chapter is an assessment of the practical merits of SEAL when defining user-interfaces.

For this purpose we give five user-interface definitions, each illustrating a typical application area. Each definition is first briefly introduced after which its SEAL script is presented, the resulting programming environment is shown, and the practical merits of SEAL in this application area are discussed. The definitions presented can also be used for tutorial purposes.

Chapter overview

Section 5.2 contains a "classical" programming environment consisting of a syntax-directed editor, a typechecker, and a compiler. Section 5.3 describes an environment for the λ-calculus with as main topic "program transformations". Section 5.4 illustrates an environment for a programming language featuring input from and output to a simulated terminal. An example of an environment used for simulating parallelism is presented in Section 5.5. Our final application area, an environment computing import relations, is presented in Section 5.6. Section 5.7 lists achievements and limitations. In Section 5.8 we sketch the future development of SEAL. Finally, we list conclusions in Section 5.9.

5.2 Simple programming environment

Our first case study is a simple programming environment consisting of a syntax-directed editor, a typechecker, and a compiler for a language L. The first enables the user to manipulate L-programs, the second checks L-programs for static semantic errors, and the third compiles L-programs to an intermediate language.

Consider a computational component generated by the ASF+SDF Meta-environment consisting of a syntax-directed editor for L, a `typecheck` function, and a `compile` function. Let these functions be defined as:

```
typecheck (Program) -> Error-list
compile   (Program) -> Intermediate-code
```

Both functions must be "connected" to the user-interface of the L-editor in the resulting interactive programming environment. The tree residing in the L-editor is the *argument* of both functions, the *result* of the `typecheck` function is to be presented in a window as feedback to the user, whereas the *result* of the `compile` function is to be written to a file.

The notions of windows or files do not exist in SEAL, editor instances must be used instead. As a consequence, SEAL uses the *same* name for both the editor and its corresponding file. Thus, presenting the result of a function in a window amounts to creating an editor instance with arbitrary name. Furthermore, writing the result of a function to a file named F amounts to creating an editor named F.

Writing a SEAL script to obtain the environment sketched above requires several steps. First, we must ask ourselves which functions should be connected to the user-interface, where do their arguments come from, and how to handle their results. These questions are already answered above. Second, we must ask ourselves what the enabling conditions for both functions are. Clearly, these conditions are equal: the tree in the

```
Configuration for language L is
button Typecheck
when focus root is Program
enable
  Prog := focus root;
  create("Typecheckerrors", L-tc : typecheck(Prog))
doc : "typecheck an L program"
```

Figure 5.1: SEAL script for a button typechecking L programs.

L-editor must be a "Program". Finally, because ASF+SDF is a modular specification formalism, we need to indicate the context (i.e. an ASF+SDF module) in which a computation should be performed. If the `typecheck` function is defined in module `L-tc` the context is `L-tc`.

We are now in a position to discuss the SEAL script (cf. Figure 5.1) for adding a `Typecheck` button to an L-editor The `Typecheck` button is only enabled if the focus tree, when moved to the root, is of sort `Program`. When the user of an L-editor presses this button, the whole tree is assigned to the SEAL variable `Prog`. Next, `typecheck(Prog)` is computed in the context of `L-tc`. A new editor instance named `Typecheckerrors` is created to show the result. The definition of the `Compile` button is similar and therefore omitted.

5.2.1 Using computed editor names

In the example above, a *fixed* named is used for the editor instance displaying the result of a function call. SEAL also allows the use of a *computed* name for an editor. This feature is for instance used in the script for the programming environment for SEAL itself[1], a part of which is shown in Figure 5.2. An impression of the resulting programming environment for SEAL is given in the Figures 5.3 through 5.5. Here, the function `outputname` maps the language name —SEAL— to "SEAL.seal.ll". When the `Compile` button is pressed, an editor is created that uses a file with that name. However, we would prefer to use the name of the editor to which the `Compile` button was added as argument, but SEAL lacks the primitives to obtain it. Moreover, the file corresponding to the created editor is always created in the current directory, or in a fixed directory when `outputname` yields a full path name.

[1]Here, we use the *previous* version of the implementation of the SEAL environment. That is, the interpreted version, not the compiled version as described in Section 4.5.5.

```
Configuration for language SEAL is
button Compile
when focus root is SEAL-spec
enable
  Script := focus root;
  Name := SEAL-Compiler : outputname(Script);
  create(Name, SEAL-Compiler : compile(Script))
doc : "compile a SEAL script"
```

Figure 5.2: Part of the SEAL script for the SEAL language.

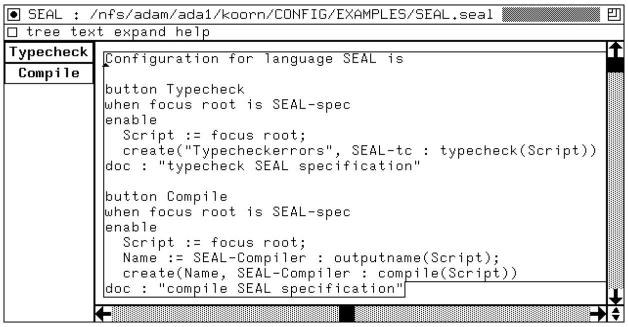


Figure 5.3: A SEAL editor containing its own script.

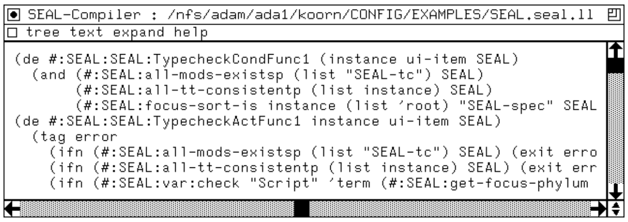


Figure 5.4: The result of compilation.

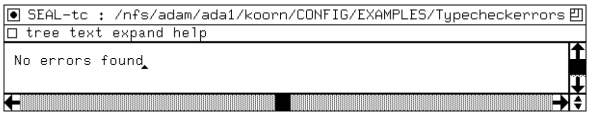


Figure 5.5: The result of typechecking.

5.2.2 Discussion

Connecting functions offered by the computational component to user-interface objects is straightforward. The obligation to use an editor instance, instead of a file, to save the result of a compilation is inconvenient. Furthermore, the mechanism in SEAL to use computed filenames lacks a primitive to obtain the name of the editor to which a user-interface object is attached. To solve these problems we suggest to:

- extend the `create` statement such that it only creates a *file* when indicated; and

- add a primitive that yields the name of the current editor.

5.3 Program transformations

Our second case study illustrates a concept frequently found in interactive programming environments: *transformations*. A transformation replaces the program, or a part of it, by an equivalent one. In many cases such a transformation is based on a conditional rule stating that a program part matching the left-hand side of the rule may be replaced by the right-hand side of the rule or vice versa. The condition and/or the replacement may also require additional information. For an overview of program transformations we refer to [Par90].

As a typical example, we will consider an environment for the λ-calculus [Gor88, Bar84]. The essential part of the SEAL script[2] for that environment is given in Figure 5.6. An impression of the resulting environment is shown in the Figures 5.7 through 5.10. We adapted the SEAL script from [Deu92] by adding an `Undo` facility and by changing the initialization procedure. Buttons for η conversion, α conversion, and left-most reduction of a λ-expression are similar to the button for β reduction and are therefore omitted. The script illustrates four concepts: program transformation without using external information (`Beta` button), program transformation using external information (`Expand` button), undoing a transformation (`Undo`), and initializing external information (`Init` button). These concepts are discussed below.

[2]We use this script here with permission of Arie van Deursen, CWI, Amsterdam, The Netherlands.

```
button Beta
when focus is L-EXP  and Convert : is-beta-redex(focus)
enable
  FocusVar := focus; create("Undo", Undo : save-l-exp(FocusVar));
  focus := Convert : beta(FocusVar)
doc: "Perform one beta reduction, if possible"

button Expand
when focus is L-EXP and Let : is-expandable(focus)
enable
  FocusVar := focus; create("Undo", Undo : save-l-exp(FocusVar));
  LetDefs := "Definitions" . focus root;
  focus := Lambda : expand(FocusVar, LetDefs)
doc: "Expand a lambda-expression according to its Let-definitions"

button Undo
when focus is L-EXP and "Undo" . focus is Saved-L-EXP
enable
  UndoVar := "Undo" . focus down;
  FocusVar := focus; create("Undo", Undo : save-l-exp(FocusVar))
  focus := UndoVar;
doc: "Undo any transformation."

button Init
when "Input" . focus root, down is Unix-filename
enable
  File := "Input" . focus root, down;
  create("Help2", Undo, readfile(File), LET);
  LetDefs := "Help2" . focus root;
  create("Definitions", LetDefs);
enable
  create("Help1", Undo,
          "The file containing your definitions is: <Unix-filename>",
          Unix-filename );
  File := select("Help1", Unix-filename);
  FileInput := "Help1" . focus root;
  create("Input", FileInput);
  create("Help2", Undo, readfile(File), LET);
  LetDefs := "Help2" . focus root;
  create("Definitions", LetDefs);
doc: "Read in a number of Let-definitions"
```

Figure 5.6: Part of the SEAL script for the Lambda language

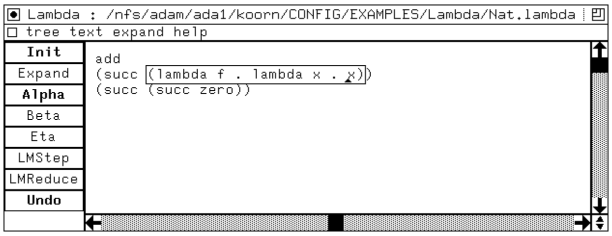


Figure 5.7: A Lambda editor containing a representation of 1 + 2.

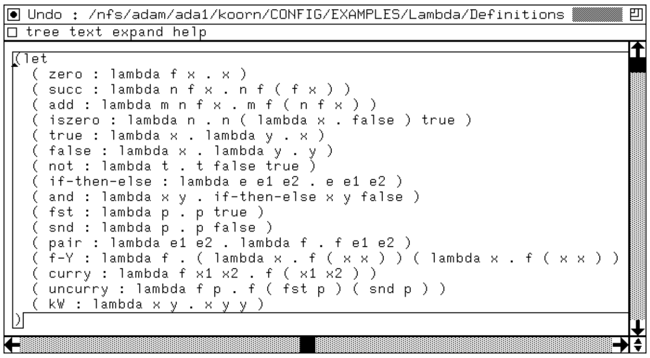


Figure 5.8: An editor containing the “let definitions”.

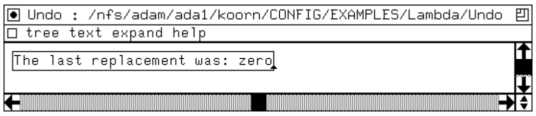


Figure 5.9: The editor containing the last replacement.

Figure 5.10: The editor containing the name of the file read to obtain “let definitions”.

5.3.1 Local transformations

A *local* transformation is a program transformation which does not need any external information. The button `Beta` in Figure 5.6 invokes such a transformation. It applies a β conversion to the expression in the focus. This conversion simulates evaluation of a function in the λ-calculus, i.e., it states that expressions like `(lambda V.E1)E2` are equivalent to `E1[E2/V]`. Here, `V` is a variable and `E1[E2/V]` represents `E1` with all occurrences of `V` replaced by `E2`.

The enabling condition for the `Beta` button amounts to checking that the expression in the focus is a λ-expression of the form `(lambda V.E1)E2`. In the SEAL script this is expressed by a check that the focus is of sort `L-EXP` —λ-expression— and by calling the function `is-beta-redex` in module `Convert` which determines whether or not the focus is of the proper form.

The action for the `Beta` button consists of two tasks. First, the expression in the focus is copied and saved in an external editor for the purpose of undoing this transformation. Next, the actual transformation is performed by assigning the result of calling `beta` in module `Convert` to the focus. The actual β conversion is performed by this function call.

5.3.2 Context-dependent transformations

A *context-dependent* transformation uses external information. An *expansion* in the λ-calculus is an example of such a transformation. It uses *let* constructs, which can be viewed as shorthands for λ-terms. Let constructs are used to represent all kinds of mathematical objects. For example, in Church's classical work [Chu41], a natural number N is represented as `lambda f x . f`^{N} `x`. A way to obtain this is by defining:

```
(let (zero: lambda f x . x)
     (succ: lambda n f x . n f (f x)))
```

According to these definitions, `succ(succ zero)` may be β reduced to `lambda f x . f (f x)`. However, before β reduction can take place, each name must be replaced by its definition in the corresponding `let` definition.

The `Expand` button implements such name expansions. Its condition uses the function `is-expandable` which searches for expandable names in the focussed expression. The action of the `Expand` button uses the `let` definitions found in *another* editor instance called `Definitions`. In this way the definitions may be changed by the user at run-time. The expansion itself is implemented by calling the function `expand` with the focus expression and the definitions as arguments. This function can not be called if there are no definitions, i.e., if the editor called `Definitions` is not present. Although a check for its existence lacks in the script, the need for it is caught by the SEAL compiler which adds it at compile time.

5.3.3 Undoing transformations

The result of a transformation may not be what the user expected. It is therefore convenient to have an *undo* facility. The script in Figure 5.6 provides a one-step undo. Before invoking a transformation, the focus expression is saved in an editor instance called `Undo` by using the `create` statement. This statement creates an editor instance when it is not yet present, otherwise it is re-used. The `Undo` button's action simply swaps the focus expression and the saved expression. For aesthetic reasons, the text corresponding to the λ-expression in the `Undo` editor is preceded by "`The last replacement was:`" (cf. Figure 5.9). This string in combination with the saved λ-expression forms a term of sort `Saved-L-EXP` where the saved expression is its first child.

Based on this one-step undo it is easy to extend the mechanism to an N-step undo by saving all transformed expressions in a *list*. If the `Undo` button is invoked, the last item in the list replaces the current expression and the item is deleted from the list.

5.3.4 Initializing external information

Expansions use `let` definitions that may be changed by the user dynamically. These definitions must therefore be saved in a file for later use. Furthermore, several files may contain such definitions. We also want to allow the replacement of the currently used definitions by definitions saved in a file. This implies that the file containing definitions may be changed at run-time. The `Expand` button uses definitions residing in an editor using a *fixed* file called `Definitions`. We therefore need a mechanism to ask the user to give a filename, to read this file and copy its contents to `Definitions`. This functionality can be implemented using SEAL as well, see the `Init` button in Figure 5.6.

`Init` is an example of a SEAL button with two condition-action pairs. Invoking the button leads, by definition, to the execution of the action associated to the first condition that holds. In order to enable the user to change the filename we save the "current filename" in an editor instance

`Input`. This editor does not exist at system start-up time, but it does exist after the first invocation of `Init`.

Consider the situation where the `Input` editor instance does not yet exist. The condition of the first condition-action pair of `Init` can not be evaluated and, by definition, fails. The condition of the second pair is empty and, again by definition, succeeds. In this situation we have to ask the user to provide a filename, read the file, copy it into the `Definitions` editor, and, as a side effect, create the `Input` editor for later use. Questions like "Please give a filename" are implemented in SEAL by using the `select` statement in combination with a *temporary* editor instance. Such editors are created by using the `create` statement with four arguments: `Name`, `Module`, `Text`, and `Sort`. It creates an editor named `Name` that uses the syntax defined by module `Module`. The editor's text is `Text` and parsing it should yield a tree of sort `Sort`. The text argument may be one or more strings or a call to another primitive, `readfile`. This primitive has one argument, either a string or a variable, indicating the name of the file to be read. Implementing the described functionality in SEAL is done by taking the following steps. Create a temporary editor `Help1` containing a placeholder for a filename. Let the user fill in that placeholder and assign the result to the variable `File`. Copy the contents of `Help1` to the permanent editor `Input`. Create a temporary editor `Help2` that uses the text resulting from `readfile(File)`. Copy the contents of `Help2` to the permanent editor `Definitions`.

Now consider the situation where the editor instance `Input` exists as a result of an earlier invocation of `Init`. The tree residing in `Input` has a subtree representing a filename as the first child of the root, so now the condition of the first condition-action pair of `Init` succeeds. Note that it also succeeds if the user has *changed* the filename at some moment after the first invocation, unless it is syntactically incorrect. If syntactically correct, we obtain the filename by inspecting the first child of the whole tree in `Input`. Here, the user has already indicated which file containing let definitions to use so we proceed with reading and copying it to `Definitions` as described above.

5.3.5 Discussion

We have shown that both local and context-dependent transformations can be implemented in SEAL in a straightforward manner. Adding a one-step undo, or even an N-step undo, is straightforward as well. However, initializing information to be used in a context-dependent transformation is rather inelegant for the following reasons:

- the same sequence of four statements appears in both actions of `Init`;
- reading a file can only be achieved by using the statement for creating temporary editor instances;
- the name of the last file read is global state information which can only be saved in an editor instance and the corresponding window is therefore constantly on the screen; and
- the necessity to use temporary editors in combination with place-holders and the `select` statement just for asking a filename is rather baroque.

To solve these problems we suggest to extend the SEAL language as follows:

- provide a macro or procedure mechanism to prevent repetition of identical sequences of statements;
- introduce a notion of global variables together with a mechanism to save their value rather than creating a new editor instance for that purpose;
- introduce a keyword `browser`, resulting in a user dialogue that uses a file-browser to yield a filename, which may be used as an argument of the `readfile` primitive; and
- introduce a `parse(Module, Text)` primitive where `Text` may be zero or more strings.

With these improvements, reading a file of the user's choice and copying it in an editor may be expressed as:

```
create("Definitions", parse(Let, readfile(browser)))
```

This single statement could then replace *all* statements used in the `Init` button. For the name of the external file containing let definitions, used in the `Expand` button and set in the `Init` button one could use:

```
global variables: File        %%declare File as a global var. in Init
File := parse(Names, browser) %%Names defines the syntax of filenames
focus := Lambda : expand(FocusVar, File . focus root) %%in Expand
```

5.4 Interactive input and output

An environment for programming languages featuring input and output (I/O) is the subject of our third case study. Generating an environment for these languages from an ASF+SDF specification alone is impossible because there are no I/O primitives in ASF+SDF. An elegant solution is to model program execution by the computational component generated from an ASF+SDF specification and to handle I/O by SEAL. This solution illustrates two interesting aspects: the *alternation* of ASF+SDF computations and performing I/O, and *validating* user input.

This behavior may be implemented by a conditional loop: execute the program in ASF+SDF until I/O is needed, take care of the I/O in SEAL, continue execution, ..., until the program terminates. The switch from executing the program to performing I/O is modeled by an *environment* containing among others the *execution-status*. This execution-status indicates whether the program is running, needs input, needs output, or has terminated.

Validating the user input is necessary to make sure that the input offered to the computational function is syntactically correct. For example, if the program needs a number as input, it should be checked that the text entered is indeed a number. If it is not, the user should be prompted again until it is. This is obtained by combining an ASF+SDF function for computing validity, SEAL statements for prompting the user and obtaining input, and a conditional loop.

The SEAL script for adding a button to invoke program execution (`Eval`) and to initialize the environment is shown in Figure 5.11. An impression of the resulting environment is shown in the Figures 5.12 through 5.14. Within the action of the `Eval` button of Figure 5.11 we decide what to do next — terminate, perform output, or perform input — by inspecting the execution-status stored in the environment. The environment is modeled by an external editor with name `Environment` which is initialized with execution-status `running`. Furthermore, user input is modeled by asking to fill in a *string* for which we use a `select` statement in combination with a `<STRING>` placeholder, which is saved in the environment for re-use. The SEAL statements used in the `Initialize` button are similar to what we encountered in Section 5.3.4 and discussion of this button is therefore omitted. Below we discuss modeling output, modeling input, and finally the complete script.

```
Configuration for language L2 is

button Eval
when focus root is PROGRAM and "Environment" . focus root is ENV
enable
  Env := "Environment" . focus root;
  Prog := focus root;
  Text := L-eval : init-output();
  while not L-eval : terminated(Env) do
    Env := L-eval : eval-until-io(Prog, Env); %%eval until I/O
    Continue :=  L-eval : make-true();
    while L-eval : output-needed(Env, Continue) do %% output needed?
      Text := L-eval : add-output(Text, Env);
      create("Terminal", Text);
      Env := L-eval : output-added(Env);
      Continue := L-eval : make-false()
    od;
    Continue := L-eval : make-true();
    while L-eval : input-needed(Env, Continue) do %% input needed?
      Text := L-eval : add-input-prompt(Text, Env);
      create("Terminal", Text);
      UserInput := select("Terminal", STRING);
      Text := L-eval : update(Text, UserInput);
      create("Terminal", Text);
      while not L-eval : input-is-ok(Env, UserInput) do
        Text := L-eval : add-error-input-prompt(Text, Env, UserInput);
        create("Terminal", Text);
        UserInput := select("Terminal", STRING);
        Text := L-eval : update(Text, UserInput);
        create("Terminal", Text) od;
      Env := L-eval : input-added(Env, UserInput);
      Continue :=  L-eval : make-false()
    od
  od
doc: "Execute a program"

button Initialize
when focus root is PROGRAM
enable
  Program := focus root;
  create("Stringmeta", L-eval, "<STRING>", STRING);
  String := "Stringmeta" . focus root;
  create("Environment", L-eval : make-env(Program, String));
doc: "Initialize the environment"
```

Figure 5.11: SEAL script for the L2 language

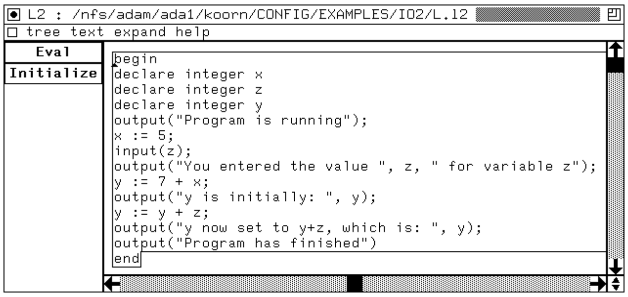


Figure 5.12: The editor containing the program.

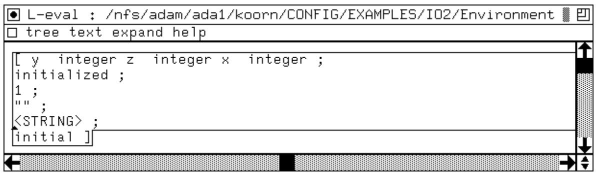


Figure 5.13: The editor containing the "environment".

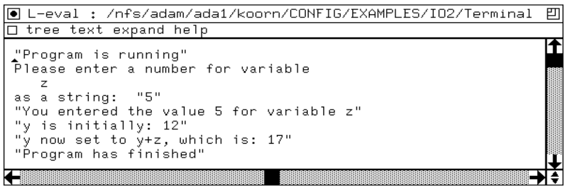


Figure 5.14: The editor containing the "terminal".

5.4.1 Modeling output

The output is modeled as a *text* —list of strings— displayed in a separate editor instance called `Terminal`. Whenever output is needed during the execution of the program, the evaluation function `eval-until-io` changes the execution-status in the environment and puts the string to be written to the terminal in the environment. In the action of the `Eval` button the function `add-output` is called with the current text and the environment as arguments. It updates the text displayed in `Terminal` and resets the execution-status to `executing` by calling the `output-added` function.

5.4.2 Modeling input with validation

User input from the "terminal" is modeled by adding a special string to the text. This string contains the *prompt* and a placeholder to be filled in by the user. The prompt may for instance be "`Fill in a number:`". Analogous to handling output, the `eval-until-io` function takes care of changing the execution-status. Furthermore, it sets up the prompt in the environment in the same way as a string that has to be written to the "terminal". The `add-input-prompt` function concatenates the prompt and the placeholder and adds the result to the text. Next, we use the `select` statement to ask the user to fill in the placeholder. The semantics of `select` is such that the user is asked to point at *any* subtree of sort `STRING`. It is thus not certain that the placeholder has been filled in, but in any case, the variable `UserInput` contains the selected string. To make sure that the text in the "terminal" displays the selection result, we replace the [prompt, placeholder] pair by a [prompt, selection-result] pair in all cases. This is implemented by the `update` function.

After the user has entered a string, we must validate that it is well-formed, e.g., that it is indeed a number. Validation itself is done using a call to the `input-is-ok` function. If it is not valid, the user is prompted again, using an "error prompt" until it is. Such a prompt may e.g. be "`... is not a number. Fill in a number:`" where "`...`" is the previously supplied string. Validation is implemented by using a *while* loop in combination with a call to `add-error-input-prompt`[3].

Finally, the necessary input value, stored in the variable `UserInput` is copied into the environment and the environment's execution-status is set to `executing` by a call to the `input-added` function.

---

[3]In Section 5.5.2 we show an alternative approach to validating user input

5.4.3 Modeling the terminal and the environment

After having modeled I/O, we discuss the remainder of the `Eval` button. The condition of `Eval` is a conjunction of "The program must be of sort PROGRAM" and "the environment must exist". The latter is implemented by checking that the `Environment` editor instance is of sort ENV, i.e., an environment. The action of `Eval` initializes the text to the empty text, by calling `init-output`, and enters a conditional loop. Next, we execute the program, by calling `eval-until-io`, until I/O is needed. Here we arrive at a point where we have to decide whether I/O is needed or not. In the latter case, the execution status is `terminated`. If I/O is needed, the environment is inspected and we decide to perform input or output.

SEAL lacks an `if-then-else` statement we are therefore obliged to use, or rather mis-use, the `while` statement for this purpose. In both cases where we use `while` as an `if` statement we use the variable `Continue` to prevent looping. It is set to true or false by calling `make-true` or `make-false` respectively.

After handling I/O if necessary, we return to the conditional loop which inspects the execution-status in the environment. If it is `terminated` the loop is exited.

5.4.4 Discussion

Generating a programming environment for a language featuring I/O is possible using SEAL. Simulating a terminal is straightforward for both output and user input except for the necessary text update after user input due to the semantics of the `select` statement. We have shown in two cases that conditional looping is a sufficiently strong concept: in the "execute; i/o; execute;..." loop and in case of "repeat asking user input until the answer is valid". However, initializing external information —the environment— is inelegant for reasons already mentioned in Section 5.3.5. Clearly, the use of the `while` statement as an `if-then-else` is very inelegant. Furthermore, the number of statements used in the action of `Eval` is rather large making it hard to read. As already stated in Section 5.3.5, it indicates the need for abstraction through macro's or procedures.

Another interesting point is the role of the `Text` variable. Each time a string is added, it is displayed using the `create` statement. The text could also be made part of the environment in which case the `Terminal` editor instance must be defined as a *view* on the text field of the environment. This approach would shorten the number of statements considerably and thus enhance readability. For example, if the environment is implemented as a global variable `Env` and the text is its first child we could use:

```
create("Terminal", Env . down)
```

To solve the problems mentioned we suggest to improve SEAL such that:

- an `if-then-else` statement is available; and
- `create` allows viewing a part of a variable.

5.5 Simulating parallelism

A programming environment for a specification language for parallel systems, such as PSF [MV90], LOTOS [ISO87] and μCRL [GP91] is our fourth case study. We present an environment for LOTOS featuring a *simulator*.

In this type of languages the notions of *events* and *choice* play an important role. As an example, consider a specification of a machine vending coffee or tea. A person, approaching the machine for a drink, has to make a *choice*: insert a nickle, a dime or a quarter. Actual insertion of a coin is called an *event*. If tea is cheaper than coffee and the sum of all coins inserted exceeds or equals the price of tea, the person may have another choice as well: give me tea, or, insert more coins until the price of coffee is reached. Note that after reaching the price of tea, the machine also has a choice besides accepting coins: it may now serve tea.

A simulator is used to study the behavior of a specified system. In the example above this might amount to test whether the machine is willing to serve tea after inserting enough coins. If not, the behavior of the machine is found incorrect. Note that at any moment in time a number of events may occur. Initially, only the insertion of a nickle, a dime, or a quarter is possible and one of these must be chosen. A simulator models this by presenting a set of possible events to its user: $\{nickle, dime, quarter\}$. The simulator user now makes a choice and the selected event is then processed by the simulator resulting in a new set of possible events. After "inserting" sufficient coins this set may become $\{nickle, dime, quarter, tea\}$ and selecting *tea* then models the machine serving tea. After each event the simulated system changes its state: initially no coins have been inserted, after a *nickle* event we arrive in a state indicating "total amount of money now inserted is 0.05$", etc. After a *tea* or *coffee* event the system may be in a state encountered earlier. In general, parallel systems lead to a graph of states called the *process graph* in which each state is a *node*.

```
Configuration for language LOTOS is

button Simulate
when focus root is SPECIFICATION
enable
  create("help", Sim_Interface, "<VALUE_EXPRESSION>", VALUE_EXPRESSION);
  ValueExprMetaVar := "help" . focus root;
  LotosSpec := focus root ;
  SimVar := Sim_Interface : createsimobj(LotosSpec, ValueExprMetaVar);
  create("object.sim", SimVar);
  MenuView := Sim_Interface : view-menu(SimVar);
  create("menu", Simulator : donothing(MenuView));
  create("node", Sim_Interface : view-node(SimVar));
  create("trace", Sim_Interface : view-trace(SimVar))
doc : "Create Simulator object from a LOTOS specification."
manual entry : LOTOS
```

Figure 5.15: SEAL script for the LOTOS language

Part of the simulator are: a window displaying the set of possible choices, a window displaying the current node in the process graph, and a window containing all choices made so far, the *trace*. The relevant parts of the *two* SEAL scripts implementing a simulator for LOTOS specifications are given in Figures 5.15 and 5.16. An impression of the resulting environment is shown in Figures 5.17 through 5.20. They are adapted from [KJT+93][4] in the sense that we added a trace window, changed the initialization procedure and simplified the undoing of events. The scripts illustrate two concepts: using more than one language and validation of user input in the computational component. These concepts are discussed below.

5.5.1 Using multiple languages

The environment for the LOTOS simulator uses *three* languages: `LOTOS`, `Simulator`, and `Sim_Interface`. The specification to be simulated is a "program" written in `LOTOS`. Therefore, editors using the `LOTOS` language should have a button which creates the simulator. A SEAL script for such a button (`Simulate`) is thus a configuration for the `LOTOS` language

[4] We use this script here with permission of Han Joosten, PTT-research, Groningen, The Netherlands.

```
Configuration for language Simulator is

button Reset
when "object.sim" . focus root is SIMOBJ
     and Sim_Interface : go_up_possible("object.sim" . focus root)
enable
  SimVar := "object.sim" . focus root;
  SimVar := Sim_Interface : goto_root(SimVar)
  create("object.sim", SimVar);
  MenuView := Sim_Interface : view-menu(SimVar);
  create("menu", Simulator : donothing(MenuView));
  create("node", Sim_Interface : view-node(SimVar));
  create("trace", Sim_Interface : view-trace(SimVar))
doc : "Start simulation at the root again"

button StepBack
when "object.sim" . focus root is SIMOBJ
     and Sim_Interface : go_up_possible("object.sim" . focus root)
enable
  SimVar := "object.sim" . focus root;
  SimVar := Sim_Interface : go_up(SimVar));
  create("object.sim", SimVar);
  MenuView := Sim_Interface : view-menu(SimVar);
  create("menu", Simulator : donothing(MenuView));
  create("node", Sim_Interface : view-node(SimVar));
  create("trace", Sim_Interface : view-trace(SimVar))
doc : "Undo the last invoked event"

button Step
when "object.sim" . focus root is SIMOBJ
enable
  TheAction := select("menu", ITEM-VIEW);
  SimVar := "object.sim" . focus root;
  SimVar := Sim_Interface : process-input(SimVar, TheAction);
  create("object.sim", SimVar);
  MenuView := Sim_Interface : view-menu(SimVar);
  create("menu", Simulator : donothing(MenuView));
  create("node", Sim_Interface : view-node(SimVar));
  create("trace", Sim_Interface : view-trace(SimVar))
doc : "Do an event"
```

Figure 5.16: SEAL script for the Simulator language

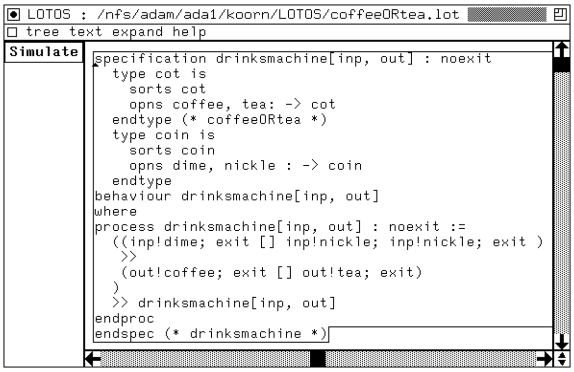


Figure 5.17: The LOTOS editor containing a specification of a machine vending coffee or tea.

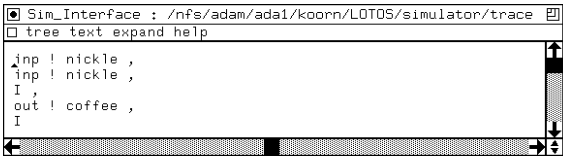


Figure 5.18: The trace window showing two nickles have been inserted, an "internal step" is made corresponding to an "exit" in the specification, coffee has been served, and an internal step is made.

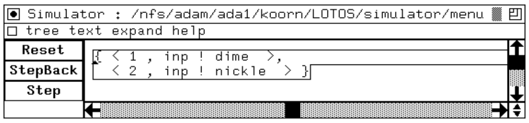


Figure 5.19: The editor containing the "menu". Possible choices are inserting a nickle or inserting a dime.

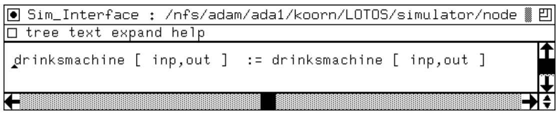


Figure 5.20: The editor containing the "node". The simulated process is about to re-start.

(cf. Figure 5.15). This button creates what is called a *simulator object*, a data structure among others containing information derived from the LOTOS specification and a <VALUE_EXPRESSION> placeholder to be used for user input later on. The created object is saved in an editor instance named `object.sim`. Three editors are created each containing a view on the simulator object: `menu` (cf. Figure 5.19) used for the set of possible events, `node` (cf. Figure 5.20) displaying the current node in the process graph, and `trace` (cf. Figure 5.18) showing the list of chosen events.

The `menu` editor displays the set of possible events and it is therefore equipped with a `Reset` button, a `StepBack` —undo last event— button, and a `Step` —do event— button. Functions connected to these buttons all relate to the menu since only the menu shows the possible events. These buttons must therefore not be added to the user-interface of any other editor. However, SEAL adds buttons to *each* editor using a particular language. The only way to prevent the addition of these buttons to the `node`, `trace`, and `object.sim` editors is to use a different language for them. Therefore, the `menu` editor uses the `Simulator` language for which a separate SEAL script is written (cf. Figure 5.16). The remaining editors, i.e., `object.sim`, `node` and `trace`, use the `Sim_Interface` language for which there is no script. They are thus not equipped with any button.

Although we obtain the desired result by using a different language for the `menu` editor, it is not straightforward to implement it. This is due to the way SEAL determines the language to be used by the editor. For example, a SEAL statement like:

```
create("node", Sim_Interface : view-node(SimVar))
```

leads to the creation of an editor named `node` using the language defined by the ASF+SDF module `Sim_Interface`. This language is thus derived from the context used for the computation. In our scripts, *all* computations, such as creating `object.sim` and computing views, are defined in that module. For each editor used it is the language we intended, `Sim_Interface`, except for `menu` which should use `Simulator`. We must therefore either move the computation of the `menu` from the ASF+SDF module `Sim_Interface` to the module `Simulator` or add an identity function to the `Simulator` module. The latter option has been chosen and the identity function is called `donothing`.

5.5.2 Validating user input

Validating user input was discussed earlier in Section 5.4.2. There, the `while` statement of SEAL was used in combination with a boolean function for checking validity. The LOTOS simulator validates user input quite differently. It does not process invalid input at all.

User input plays a role when an event contains *data*. In LOTOS one does not only define parallel processes but also the exchange of data between processes. Modeling data as processes easily leads to an infinite process graph, e.g., when natural numbers are used. In a description of a system's behavior one therefore uses a *datum* of some *sort* to keep the graph finite. Within the simulator the exchange of a datum implies user input of the datum and checking that it is of the proper sort. The functionality described above is implemented in SEAL as follows.

When the simulated system is in a state where "exchange a datum" is a possible event, the `menu` displays an event containing the placeholder `<VALUE_EXPRESSION>`. This placeholder models the datum and was created in the action of the `Simulate` button (cf. Figure 5.15). The `Step` button (cf. Figure 5.16) uses SEAL's `select` statement to ask the user a choice from the `menu`. If an event without a placeholder is selected, the event is processed by calling the function `process-input` yielding an updated simulator data structure. If an event containing a placeholder is selected however, SEAL's `select` does not return unless the placeholder is filled in by the user. Eventually, the selected event is assigned to the `TheAction` variable and `process-input` is called. The filled in placeholder is now validated within `process-input`. It returns an updated simulator data structure if the input is found valid otherwise it returns the current simulator data structure. In the latter case, exactly the same `menu`, `node`, and `trace` are computed because the simulator data structure is left unchanged.

5.5.3 Discussion

We have shown how an environment for simulating parallelism can be obtained using SEAL. By using more than one language an environment can be built where specific editor instances are supplied with specific buttons. However, the introduction of an identity function in a specific ASF+SDF module was required. The source of this problem is the way SEAL determines the language to use for an editor from the context module of a computation as in:

```
MenuView := Sim_Interface : view-menu(SimVar);
create("menu", Simulator : donothing(MenuView))
```

However, one might also argue that the source of the problem is quite different: SEAL adds user-interface objects defined in a script for language

L to *all* editors parameterized with L. Solutions of the problem could be to add an optional argument to the `create` statement to explicitly express which module to use. An alternative solution is to add an argument to `create` explicitly expressing whether or not SEAL should add user-interface objects to the created editor.

Validating input may be obtained solely by using the computational component. This may be used as an alternative for the validation mechanism described in Section 5.4.2. We prefer the mechanism described there for two reasons. First, it provides feedback to the user which might contain an explanation *why* a given value was incorrect. Second, the mechanism used here leads to a re-computation of the menu, the node, and the trace *each* time the input is invalid. This is prevented when the mechanism described in Section 5.4.2 is used.

To solve the problems mentioned we suggest to improve SEAL as follows:

- add an optional argument to `create` which explicitly express which module to use, or,

- add an optional argument to `create` which explicitly express whether or not the addition of user-interface objects is required.

5.6 Computing import relations

Our fifth and final case study is a programming environment for computing import relations. Here we present a SEAL script (cf. Figure 5.21) for a fictitious modular language we called `Imports`. Each module may import any number of other modules, circular imports are allowed as well as modules that do not yet exist. The idea is to add a button `ShowImports` to each existing module that reports to the user which modules are imported and which of those do not yet exist. The result, i.e., the transitive closure of the import relation, is displayed in a window. For the sake of simplicity, we assume that a module M is saved on a file with the same name. The resulting environment in shown in Figures 5.22 through 5.27. Furthermore, the same window is used to monitor the calculation. A screendump of that window during processing is shown in Figure 5.28. This case study illustrates two concepts: *cooperating editors* and *animation of execution.*

```
Configuration for language Imports is

button ShowImports
when focus root is Module
enable
  NamesDone := Interface : empty-set();
  NotPresent := Interface : empty-set();
  NamesToDo := Interface : empty-set();
  Module := focus root;
  NamesToDo := Interface : process(Module, NamesToDo, NamesDone);
  create("Status", Interface : processing(NamesToDo, NamesDone));
  while Interface : non-empty(NamesToDo) do
    Name := Interface : first(NamesToDo);
    File := Interface : name-to-filename(Name);
    %% if File exists, process it, otherwise, add File to NotPresent
    Continue := Interface : make-true();
    while File . focus root and Interface : eval(Continue) do
      Module := File . focus root;
      NamesToDo := Interface : process(Module, NamesToDo, NamesDone);
      create("Status", Interface : processing(NamesToDo, NamesDone));
      Continue := Interface : make-false() od;
    Continue := Interface : make-true();
    while not(File . focus root) and Interface : eval(Continue) do
      %% File does not exist, add it to NotPresent
      NotPresent := Interface : add(Name, NotPresent);
      Continue := Interface : make-false() od;
    NamesToDo := Interface : delete(Name, NamesToDo);
    NamesDone := Interface : add(Name, NamesDone);
    create("Status", Interface : processing(NamesToDo, NamesDone)) od;
  create("Status", Interface : make-imports(NamesDone, NotPresent))
doc: "Derive all imports."
```

Figure 5.21: SEAL script for the Imports language

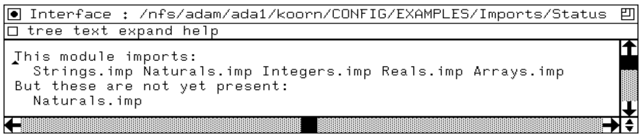


Figure 5.22: The editor displaying the import relation for the `Program.imp` module.

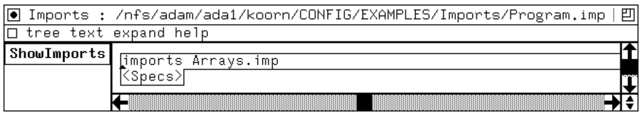


Figure 5.23: The editor containing the module `Program.imp`.

Figure 5.24: The editor containing the module `Arrays.imp`.

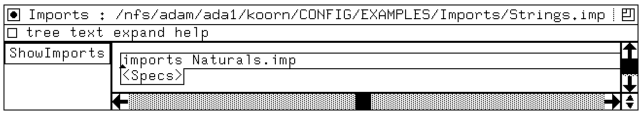


Figure 5.25: The editor containing the module `Strings.imp`.

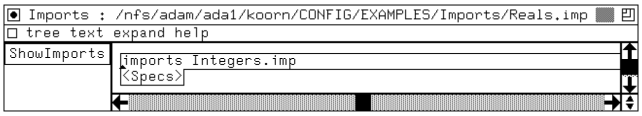


Figure 5.26: The editor containing the module `Reals.imp`.

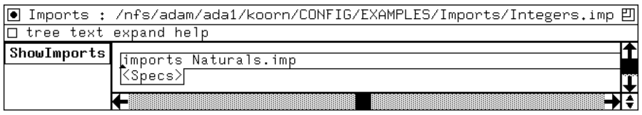


Figure 5.27: The editor containing the module `Integers.imp`.

Figure 5.28: The editor displaying the execution status during computation

5.6.1 Cooperating editors

The idea for the `ShowImports` button is as follows. We use three sets `NamesDone`, `NotPresent`, and `NamesToDo`, all initially empty, to hold module names that are respectively processed, not found, and are to be processed. We start with deriving the imported modules from the module residing in the editor to which `ShowImports` is connected. This is done by calling the function `process` which returns the union of imported module names and `NamesToDo`, but minus `NamesDone`. The result is assigned to `NamesToDo`. While `NamesToDo` is non-empty, process an element of it as follows. Assign an element to the variable `Name` and convert it to a filename. Next, test if the file exists[5]. If so, call the `process` function with the module, residing in the editor using the filename, as argument. If not, add `Name` to `NotPresent`. Finally, delete `Name` from `NamesToDo`, add it to `NamesDone` and return to the "while non-empty" loop.

After the loop, `NamesDone` contains all imported module names and `NotPresent` contains names of non-existing modules. This information may now be used in any computation on the modules e.g. creating the non-existing ones or merge all modules into one large module.

5.6.2 Animation of execution

We may animate the derivation of all imported modules by showing the sets `NamesToDo` and `NamesDone` after each update of `NamesToDo`. Each of these is preceded by a fixed string such as "`Modules to be processed:`" for readability. For this reason we call the `processing` function which result is showed in an editor instance `Status`. This editor is also used to display the final result for which we call `make-imports` with `NamesDone` and `NotPresent` as arguments.

5.6.3 Discussion

We have shown that SEAL can be used to implement a programming environment that computes import relations and we have seen how the computation process can be animated. The animation was straightforward, but for the computation process we needed an "existence check" and a loop. The check for existence was implemented by a test whether or not the focus

[5]This can only be implemented by inspecting if a focus move, e.g., to the root, is possible or not. Note that due to SEAL's identification of editor names and their corresponding file names, in combination with our assumption that a module M is saved on file M, this check equals checking the existence of M

of an editor instance could be moved to the root. Clearly, using a "move check" as an "existence check" is inelegant.

We have to use an imperative loop because SEAL lacks procedures as already noticed in Section 5.3.5. If SEAL featured a procedure call, we could have used recursion. Furthermore, we implicitly assumed that the name of an imported module is sufficient information to derive the filename, or the editor name, containing the module. This limits the applicability of the presented script. If module names are unequal to filenames and/or editor names, we need mappings from the one to the other. These mappings may be obtained if the necessary information is made explicit in yet another editor instance. Otherwise, SEAL needs to be extended with primitives providing it.

To solve the problems mentioned we suggest to improve SEAL as follows:

- add an "existence check" statement;
- add a procedure mechanism and allow recursive calls to them; and
- add primitives for mapping filenames to editor names and vice versa.

5.7 Achievements and limitations

In the preceding sections we have discussed several SEAL scripts each illustrating different aspects found in interactive programming environments. While doing so, we encountered a number of "problems" most of which were due to a lack of expressive power of SEAL. The problems could be circumvented using the current version, but only in a more or less inelegant way. We give an overview of SEAL's achievements, its limitations and we discuss these limitations at a more abstract level. These discussions are then used in Section 5.8 where we sketch SEAL's future.

The current version of SEAL has achieved the following:

- the perfect separation of the user-interface component and the computational component of a generated interactive programming environment;
- the ability to define user dialogues that can be interfaced with functions in the computational component;
- the ability to use input and output;

- the ability to use multiple, cooperating editor instances for an arbitrary task; and
- an implementation that allows full interactive development of a user-interface component.

The current version of SEAL suffers from the following limitations:

- insufficient syntax;
- a too limited interface with the computational component when information is only available in textual format; and
- a too strong relation between the notions text, file, tree, window, and editor.

We distinguish limitations at the *syntactical* level and problems at the *semantical* level. Below we discuss them both.

5.7.1 Problems at the syntactical level

We have encountered three problems at the syntactical level. First, the syntax is too strict since, e.g., arguments of a function call in an action can only be variables. Focus expressions, such as `focus` or `root`, are not allowed. On the other hand, function calls in a condition must have focus expressions as arguments. This makes sense because there is no `assign` statement in conditions and there are no global variables. Second, we have found that an if-then-else statement is missing. Third, some form of abstraction is needed to prevent repetitions of statements in different parts of a script.

5.7.2 Problems at the semantical level

There are two problems at the semantical level: interfacing with the computational component when information is only available in textual format, and, a too strong relation between the notions text, file, tree, window, and editor.

We made the design decision in SEAL that all information is available in tree format. This decision was based on the observation that functions used for computations need arguments in *tree* format. This decision causes problems for computations using information that is only available in some other format. An example of such a problem is a computation on a filename. Note that, in general, such a filename is located in the data structure of the ASF+SDF Meta-environment as a *string*. The SEAL run-time system must thus *parse* a filename after retrieving it from that data structure. Note that in the situation where unparsing is required, for instance, when creating an editor with a computed name, this problem is already solved. The SEAL run-time system is offered a filename in tree format which is unparsed before it is used, in other words, SEAL *interprets* the result of a function call.

An example of the second problem is the lack of global variables holding trees. This implies that trees can not be shared between user-interface objects. Sharing is only possible by storing these trees in editor instances. As a result superfluous windows are on the screen and, worse, changing such trees implies their superfluous unparsing.

5.8 Towards a more powerful language

The future of SEAL not only requires solving the problems mentioned in the previous section, but also involves some extensions. Generalizing these solutions and extensions would yield a far more powerful language.

5.8.1 Solving the problems at the syntactical level

Solving the problems encountered at the syntactical level is relatively simple. Clearly, the syntax used for arguments of function calls in actions must be extended and the missing if-then-else statement must be added. Preventing repetitions of equivalent statements can be obtained by the introduction of either procedures or macro's. The former solution is the most appealing since it possibly leads to *recursive* procedure calls.

5.8.2 Solving the problems at the semantical level

At the semantical level we encountered two problems and we suggest a general solution for both of them.

Our first problem is interfacing with the computational component when information is only available in textual format. This is solved by adding explicit parsing and unparsing primitives. In scripts, we should then be able to use primitives representing information located in editors or in the ASF+SDF Meta-environment. These primitives must then yield a *string*, that is, they can be used as arguments of `parse` primitives.

To illustrate that this approach increases SEAL's expressive power we present two examples. As a first example, consider a SEAL primitive filename yielding an editor's filename as a string. Parsing an editor's filename might thus look like:

```
File := parse(M, S, filename)
```

where M is the name of an ASF+SDF module and S is the name of an ASF+SDF sort. A second example is the comparison of two sorts in a SEAL condition. This is, for instance, necessary for a general Undo button for structural editing. Here, the last tree cut, say T, may be of sort $S1$ but the focus may be of sort $S2$. When Undo replaces the focus by T, it is only allowed when $S1$ equals $S2$. The button should only be enabled when this is the case, but currently it is *impossible* to express this. If SEAL would provide a primitive sort(<tree>) yielding the sortname as a string, we could use:

```
M : equal(parse(M, S, sort(T)), parse(M, S, sort(focus)))
```

Our second problem at the semantical level is a too strong relation between the notions text, file, tree, window, and editor. The discussion above may be regarded as an example of breaking the relation between text and tree. However, the text involved consisted of only one string. By allowing an arbitrary text —zero or more strings— we fully break this relation. The other relations can be broken in the same way, i.e., by introducing explicit conversion primitives. We therefore omit a description in full detail.

We give four examples of breaking the relation between text and tree. Our first example is the conversion of the contents of an arbitrary file to a tree. It may be expressed as:

```
Tree := parse(M, S, readfile(browser))
```

where browser is a primitive yielding a filename as a string, after interacting with the user using a file browser of course. The second example is creating editor placeholders, another frequently encountered problem. It can be handled by using parse(M, S, "<S>"). A third example is the execution of an arbitrary Unix command. It may be expressed as:

```
unix(<cmd>, <Text>)
```

Here, <cmd> is an arbitrary program reading from standard input to which <Text> is re-directed. <Text> may be the result of an unparse primitive and the result of execution may, for instance, be used as argument of the parse primitive. Our final example is found in cases where a computation depends on the *position* of the focus, i.e., the "path" from the root to the focus. Here, SEAL might provide a primitive path yielding a string consisting of natural numbers and spaces. After parsing this string, it can be used in any computation.

5.8.3 Future extensions

We mention two future extensions currently considered. The first is to incorporate a statement to highlight a subtree in an editor. This may be used in applications concerning animation of program execution or for user feedback after a computation in the computational component. Both are related to what we call *origin tracking*, i.e., maintaining relations between subtrees during the rewriting process [DKT93]. For example, when a program is typechecked resulting in an error message like "variable x not declared", one wants to *show* the user where "x" was used incorrectly. The second is allowing manipulations in editor instances forming the user-interface of the Meta-environment itself. That is, manipulating ASF+SDF modules themselves. This is currently not allowed due to the implications it will have on the run-time code. Note that when this would be allowed, both syntactical and semantical definitions may change *during* the execution of SEAL statements. In some cases, the result of a function call in the computational component is thus no longer valid. These cases have to be ruled out, or we need to introduce a mechanism to recompute the result.

5.8.4 Generalization

When discussing the problems encountered at the semantical level (cf. Section 5.7.2), we observed that the current version of SEAL interprets the result of a function call when a computed filename is used. In that situation, the computational component provides information —the filename– to be used by SEAL for creating an editor instance. In most other situations, however, SEAL provides information to be used by the computational component, such as arguments of functions. This observation leads to a very promising generalization.

Consider the very last example mentioned in Section 5.8.2 where a computation depended on the position of the focus. We discussed that SEAL could provide a primitive `path` which could be used in any computation. The result of a computation may then also yield a *new* path which could be interpreted by the SEAL run-time system as well. A `move` statement then moves the focus to a new position. We thus obtain *computed* focus movements and thus computed focus positions. In the current version of SEAL this can only be achieved using nested conditional loops in which a path is first built, a new path is computed and then interpreted in the script itself. However, the above suggested approach is far more elegant.

Generalizing this approach, functions in the computational component could yield SEAL statements as value, which would then be interpreted by the SEAL run-time system. Generalizing even further leads to the interpretation of complete scripts. From a research point of view this is very interesting, since it leads to a situation where SEAL scripts may be computed by ASF+SDF functions making scripts fully dynamic. It might even be possible to create and or adapt scripts recursively when we introduce the run-time interpreter as a SEAL statement.

5.9 Discussion and conclusions

Although we have encountered a number of problems in the current version of SEAL we are convinced that we are on the right track. Strong points are the predefined visual and behavioral aspects in the generated user-interface, and the perfect connection to functions in the computational component. Obtaining and manipulating information stored in the generated environment, is currently inelegant, tedious, or even impossible. To solve these problems we suggest to introduce explicit parsing and unparsing in combination with well-chosen primitives to obtain the information in a textual format. This will enhance SEAL's expressive power considerably and it will lead to more elegant scripts. Generalizing these suggestions leads to run-time interpretation of SEAL scripts which we regard as very promising. Future extensions such as highlighting subtrees and manipulating ASF+SDF modules will probably profit from this flexible and powerful approach as well.

Chapter 6

A specification of structure editing

We present an ASF+SDF specification of the structure editing part of GSE. This specification is combined with a SEAL script in which the editor commands are invoked by pressing buttons. The feasibility of using the specified editor, instead of the hand-written one, in a practical software development environment is addressed.

6.1 Introduction

In Chapter 3 we have replaced the parts of GSE dealing with text editing and user-interface management by external components, Epoch and OSF/Motif respectively. Here we investigate the possibility to *generate* the remaining —structure editing— part from an ASF+SDF specification. This leads to *simulating* the structure editing behavior of GSE. The simulation is described by an ASF+SDF specification and a SEAL script (cf. Chapter 4).

A formal definition of GSE can be used to:

- provide a better understanding of structure editors in general, and of GSE in particular;
- study the feasibility, or the implications, of future extensions to GSE; and
- bootstrap GSE by compiling its specification to C [KW93].

Studying the feasibility, or the implications, of future extensions to GSE and the wish to bootstrap the ASF+SDF Meta-environment are the main reasons for the work presented here.

The ASF+SDF specification is needed to describe both the internal state of the editor (e.g. the current tree, the current focus) and all operations provided by it (e.g. move to next child). Note that the complete editor state is itself a term which can be displayed and changed by a standard *term-editor* in the ASF+SDF Meta-environment. The SEAL script is used to extend the standard term-editor with buttons modeling the user-commands of the simulated editor.

Chapter overview

In Section 6.2 we discuss generic structure editors and their relation to language definitions. Next, we discuss structured editing in Section 6.3 and the treatment of lists —arbitrary repetitions of syntactical constructs— in Section 6.4. These discussions are formalized in Sections 6.5 through 6.6 by presenting ASF+SDF modules which define notions like legal subtree replacements and structured editing based on a placeholder/template mechanism. In Section 6.7 we describe an implementation of the simulated editor. Related work is presented in Section 6.8. Finally, we discuss the results our work in Section 6.9.

6.2 Languages, grammars, trees and signatures

A structure editor needs knowledge about the language of the programs to be edited. In particular, the tree construction rules (also known as abstract syntax) for that language have to be known by the editor. In a structure editor dedicated to a particular language, this knowledge is hard-wired in the implementation of the editor. A generic structure editor, which is our goal, must be parameterized with the language definition. A generic structure editor can be characterized as a triple of the form (*Tree*, *Path*, *Language*). A *Tree* is the abstract syntax tree to be edited, the *Path* designates the "current subtree" in the tree and the *Language* is used to check that the tree remains well-formed.

A language definition consists of two parts: *concrete syntax* defined by a context-free grammar and *abstract syntax* defined by a signature. As an example, consider the language L for expressions like `x + y * z`. The

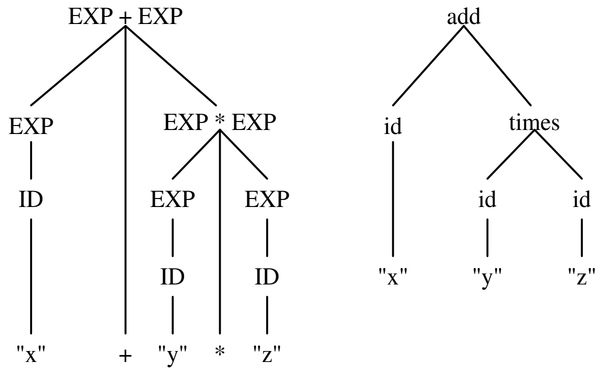


Figure 6.1: Parse tree and abstract syntax tree for `x + y * z`.

concrete syntax —at the left hand side— and the abstract syntax —at the right hand side— of L may be defined as[1]:

```
EXP ::= EXP + EXP
EXP ::= EXP * EXP
EXP ::= ID
```

```
sorts:
   ID EXP
subsorts:
   EXP > ID
functions:
   add:    EXP EXP -> EXP
   times:  EXP EXP -> EXP
   id-exp: ID      -> EXP
```

A *parser* for L maps a sentence to an abstract syntax tree in two phases. It first builds a *parse tree* using the grammar and then maps that tree to an *abstract syntax tree* using the signature. The parse tree contains superfluous nodes which do not occur in the abstract syntax tree, for example, the nodes corresponding to keywords and *chain rules*. A chain rule corresponds to a grammar rule of the form `N1 ::= N2` where both `N1` and `N2` are non-terminals. Figure 6.1 shows both the parse tree and the abstract syntax tree for the sentence `x + y * z`.

6.3 Structured editing

A structure editor uses *placeholders*, corresponding to sorts in the signature and *templates*, corresponding to functions in the signature. For example,

[1] We omitted the definition of `ID` which represents an identifier.

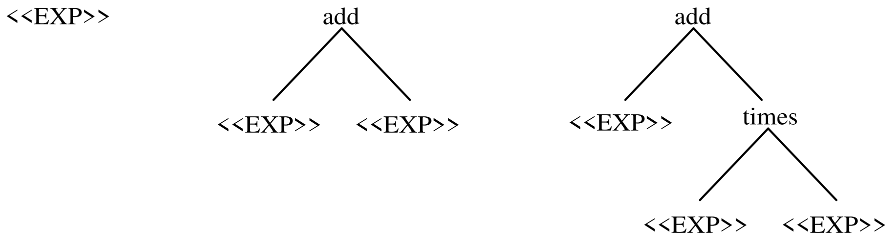


Figure 6.2: Abstract syntax trees during structure editing.

the user of a structure editor constructs a tree corresponding to the sentence `x + y * z` from an `EXP` placeholder as follows. First focus on `EXP` and replace it by the template `add` yielding a tree corresponding to the text `EXP + EXP`. Then focus on the rightmost `EXP` —again a placeholder— and replace it by the `times` template, yielding a tree corresponding to `EXP + EXP * EXP`, etc. This process is shown in Figure 6.2.

The example above shows that the user of the structure editor is offered two facilities:

- The possibility to move through the tree and reach a desired subtree.
- The replacement of the subtree focus.

We now discuss each of these facilities.

6.3.1 Focus manipulations

Moving through the tree is done by manipulating a *path*. The subtree indicated by the path is called the *focus tree* or just *focus*. The user of the editor may perform the following manipulations on the path: `root`, `up`, `down`, `next` and `previous`. The manipulation `root` means "let the focus tree be the whole tree", `up` means "go to the parent node", `down` means "go to the leftmost child", `next` means "go one child to the right" and `previous` means "go one child to the left".

Each of the manipulations will be part of the editor's set of user commands. Note that these commands may all fail, except `root`. A failing command will leave the path unchanged.

6.3.2 Focus replacements

Above we sketched how a user of a structure editor replaces the tree in focus. We formalize this notion here. Each node in the abstract syntax tree corresponds to a function in the signature. In addition, the structure editor needs nodes representing placeholders. We therefore extend the signature with special functions called *placeholder functions*. More precisely, for each sort S in a signature Σ, we extend Σ with a function (constant) of the form `<<S>>: -> S`. Placeholder functions are only needed during editing and are therefore *not* part of the language definition.

The correspondence between nodes and functions is the basis for structured editing. A structure editor only allows the construction of trees that are well-formed w.r.t. the signature. Informally, a placeholder node for sort S may only be replaced by a tree corresponding to a function of a compatible —equal or subsort— sort. These notions are defined as follows. For convenience, we assume signatures to be closed under transitivity w.r.t. the subsort relation.

Definition 1.
Let `S` and `T` be sorts in a signature Σ. `S` is *compatible* with `T` iff either `S = T` or there is a subsort declaration `T > S` in Σ. □

Definition 2.
Let T be a tree and let Σ be a signature. T is *well-formed* w.r.t. Σ iff, for each node N in T, the following conditions hold:

- F_N, the function corresponding to N is an element of the set of functions of Σ, and
- If N_i is the i-th child of N, then F_{N_i}'s sort is compatible with F_N's i-th argument sort, and
- N has K children $\Leftrightarrow$ F_N has K arguments. □

We can now easily define a *legal subtree replacement* w.r.t. a signature.

Definition 3.
Let T be a well-formed tree w.r.t. Σ, and let t be a subtree of T. A subtree replacement of t by some t' is *legal* iff

- t' is well-formed w.r.t. Σ, and
- if $t \neq T$: let t be the i-th child of its parent node P. The sort of the root of t' is compatible with the i-th argument sort of the function corresponding to P. □

We are now able to describe editing using placeholders and templates in more detail. The edit action of replacing a placeholder by a template works as follows. First, derive the sort (S) of the placeholder node. Second, compute the set $\{S_1, S_2, ..., S_n\}$ of all sorts compatible with S. Third, compute the set of functions $\{F_1, F_2, ..., F_k\}$, i.e., the union of all functions of sort S_1, all functions of sort S_2, etc. Exclude placeholder functions because these are not part of the language definition. Fourth, let the user choose an F_i. Finally, replace the placeholder by the template corresponding to F_i.

The last step requires mapping a function to a tree. This mapping inspects the signature to create a well-formed tree. The function corresponding to its root is guaranteed to be of a compatible sort due to the selection process described above.

6.4 Editing lists

Repetitions of syntactical constructs, or *lists*, are frequently found in programming languages as well as in other languages. There are three ways to describe a list in a signature: *cons* lists, binary lists, or *flat* lists.

A tree corresponding to a *cons* list consists of nodes that have two children, a list item and the rest of the list. The second child of a node corresponding to list item may also be a constant, i.e., a node without children, indicating the end of the list. Figure 6.3(a) shows a tree corresponding to a cons list of three statements. A *binary* list may be considered as a variant of a cons list: a binary function is used as list constructor. In the cons list representation this function is right-associative, whereas in the binary list representation any form of grouping may be used as long as each node has exactly two or zero children. Figure 6.3(b) depicts a tree corresponding to a binary list of three statements. Conversely, a tree corresponding to a *flat* list consists of a node that has an arbitrary number of children, where each child is a list item. Figure 6.3(c) depicts a tree corresponding to a flat list of three statements.

Flat lists are convenient in a structure editor for two reasons. First, list nodes simplify path manipulations in a structure editor. Consider, for example, a user invoking an `up` command when the focus is a list item. When using flat lists, the focus is set to the whole list immediately. When using cons lists, a focus move from item N to the whole list requires N such commands. Binary lists have a similar disadvantage. Second, and more important, signatures with list-constructor functions are more close to formalisms, such as SDF [HHKR89], that allow the definition of repetitions of syntactical constructs. Recall from Section 6.2 that a signature defines the abstract syntax while a context-free grammar defines the con-

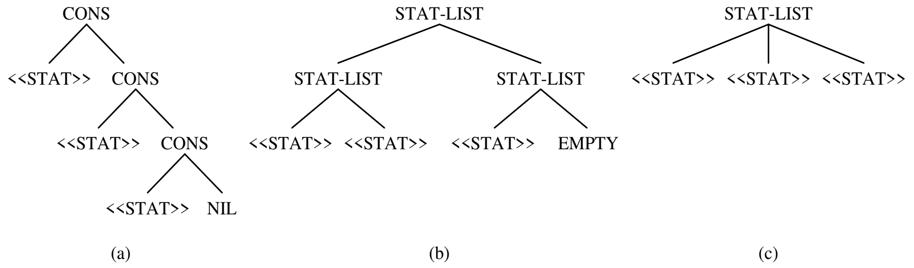


Figure 6.3: Tree representations for a list of three statements (a) cons list (b) binary list (c) flat list.

crete syntax of a language. In a context-free grammar one may define zero or more repetitions of a syntactical construct as: `LIST ::= ITEM*`. It is therefore more natural to define the corresponding function in the signature as `list: ITEM* -> LIST` rather than using a binary function and a constant.

6.4.1 Incorporating flat lists in the signature

Due to the *arbitrary* arity of nodes corresponding to flat lists we extend our signature with a *list-constructor* function. This function represents an infinite number of functions, one for each possible arity.

We use the style of SDF [HHKR89] to incorporate flat lists in the signature. All items of a list are of the same sort, the *basic sort* of the list. A list may or may not have a *separator*, an arbitrary string not containing spaces. By adding separators we stay as close to context-free grammars as possible: it offers us the possibility to distinguish different repetitions of the same basic sort. Finally, a list has an *iterator*, either '`*`' or '`+`', indicating a possibly empty list or a list containing at least one item. The list constructor function for a list sort `{S <sep>}<iter>` in the signature is of the form:

```
S-<sep>-<iter>-list: {S <sep>}<iter> -> {S <sep>}<iter>
```

Furthermore, a *list subsort declaration* of the form `{S <sep>}<iter> > S` must be added because items in the list are of a basic sort.

6.4.2 Replacing list placeholders

The introduction of lists complicates the placeholder replacement scheme sketched above. Consider, for example, replacing a placeholder `<<{S}+>>` by a template corresponding to the `S--+-list: {S ""}+  -> {S ""}+`

function. This function represents an infinite number of functions, we therefore do not know what tree should be built. We decided to build a tree with *one* child, in our example this child corresponds to the `<<S>>` function. Note that the editor must therefore distinguish between the replacement of list placeholders and the replacement of non-list placeholders.

6.4.3 List editing commands

List nodes ask for a special set of editing commands such as inserting or deleting items. Our structure editor offers two commands to insert a new item: *before* or *after* the focus. If the focus is a *list* node, the commands insert the new item before the first item or after the last item respectively. If the focus is a *list item*, the commands insert the new item before or after the focus item. In both cases the focus is moved to the newly inserted item. When the user wants to *delete* an item the focus must be the item to be deleted. However, if the list iterator is '`+`' and the item to be deleted is the only item in the list, the delete command fails.

6.5 Definition of a generic structure editor

We are now in a position to define a generic structure editor. Such a definition requires the definition of signatures, trees, the editor itself, and path manipulations. These definitions are presented in this section. Structured editing will be defined in Section 6.6.

6.5.1 Definition of signatures

Signatures were introduced in Section 6.2 and extended in Section 6.4.1. We now present their formal specification in ASF+SDF (cf. Section 4.5.1).

A *signature* consists of three sets: *sort* declarations, *subsort* declarations, and *function* declarations. The sort declarations contain all sorts that are used in the subsort and function declarations. Function declarations may have two forms. *Prefix functions* consist of a function name, zero or more argument sorts, and a result sort. For example, the `And` function of the Booleans language is represented as `And : Bool Bool -> Bool`. It expresses that `And` has two arguments of sort `Bool` and that its result sort is `Bool`. *Lexical functions* consist of a function name, one string and a result sort. Their purpose is to represent lexical entities like identifiers and numbers.

```
module Signature
imports Text
exports
sorts Signature Basicsort Sort Functionname Function Subsort
lexical syntax
  [A-Z][A-Za-z\-]*                            -> Basicsort
  [A-Za-z\-]+                                 -> Functionname
  [A-Z][A-Za-z\-]* "-" ~[ ]* "-" [\+\*] "-list" -> Functionname
  "<<" ~[ ]* ">>"                             -> Functionname
context-free syntax
  "sorts:" Sort* "subsorts:" Subsort* "functions:" Function* -> Signature
  Basicsort                                                  -> Sort
  "{" Basicsort String "}" "+"                               -> Sort
  "{" Basicsort String "}" "*"                               -> Sort
  Sort ">" Sort                                              -> Subsort
  Functionname ":" Sort* "->" Sort                           -> Function
  Functionname ":" String "->" Sort                          -> Function
variables
  Fname[']* -> Functionname       Sig[']* -> Signature
```

Figure 6.4: Relevant part of module Signature

We impose several restrictions on signatures. For example, if F is an element of a signature Σ, the result sort of F and all its argument sorts must be elements of the set of sorts of Σ. Likewise, if and only if F is a function of the form `name1: S -> T` and there is a subsort declaration `T > S` in Σ, then F is a chain function. The set of subsorts must be closed under transitivity.

The relevant parts of the module defining signatures are shown in Figure 6.4. We omitted all equations.

6.5.2 Definition of trees

A *tree* is a structured set of *nodes* and represents a *term* over a signature. A tree is a quadruple of the form (*Kind*, *Name*, *Rank*, *Children*). The *Kind* of a tree is either `tree` or `atom` for trees with or without children respectively. Other tree kinds are `list` for trees corresponding to list constructor functions and `placeh` for placeholders. The *Name* corresponds to a function name in the signature. *Children* is either a —possibly empty— list of trees or a string. In the latter case, the tree is said to be a *lexical* tree. The children are ordered from left to right and their position in the list is

```
module Trees
imports Signature Integers
exports
sorts Tree Trees Kind
context-free syntax
  "[" Kind "," Functionname "," INT ";" Tree* "]"  -> Tree
  "[" Kind "," Functionname "," INT ";" String "]" -> Tree
  Tree*                                            -> Trees
  "atom"                                           -> Kind
  "tree"                                           -> Kind
  "list"                                           -> Kind
  "placeh"                                         -> Kind
  kind(Tree)                                       -> Kind
  name(Tree)                                       -> Functionname
  rank(Tree)                                       -> INT
  children(Tree)                                   -> Trees
  child(Tree, INT)                                 -> Tree
variables
  Tree[']* -> Tree Trees[']* -> Tree* Kind[']* -> Kind
equations
[1a] child([Kind, Fname, Int; Tree Trees], 1) = Tree
[1b] Int' > 1 = true
     ===============
     child([Kind, Fname, Int; Tree Trees], Int') =
           child([Kind, Fname, Int; Trees], Int' - 1)
```

Figure 6.5: Relevant part of module Trees

indicated by a non-negative number called the *Rank*. The leftmost child will have rank 1 and the root will always have rank 0.

Consider the term `True And False`. The functions involved in this term are `True: -> Bool`, `False: -> Bool`, and `And: Bool Bool -> Bool`. The corresponding tree representation is:

```
[tree, And, 0; [atom, True, 1;] [atom, False, 2;]]
```

As an example of lexical and chain functions, consider the expression term `x + y`. The involved functions here are `id: "" -> ID` (a lexical function), `id-to-exp: ID -> EXP` (a chain function, assuming the presence of a subsort declaration `EXP > ID`) and `add: EXP EXP -> EXP`. The corresponding tree is:

```
[tree, add, 0; [atom, id, 1; "x"][atom, id, 2; "y"]]
```

The relevant parts of the module defining the tree datatype are shown in Figure 6.5.

```
module SE
imports Trees Booleans
exports
sorts SE Path
context-free syntax
  "[" Tree "," Path "," Signature "]"  -> SE
  "[" INT* "]"                         -> Path
  tree(SE)                             -> Tree
  path(SE)                             -> Path
  sig(SE)                              -> Signature
  rootp(SE)                            -> BOOL
variables
  SE[']* -> SE     Path[']* -> Path     Ints[']* -> INT*
equations
[1a] rootp([Tree, [], Sig]) = true
[1b] rootp([Tree, [Int Ints], Sig]) = false
[2]  tree([Tree, Path, Sig]) = Tree
[3]  path([Tree, Path, Sig]) = Path
[4]  sig([Tree, Path, Sig]) = Sig
```

Figure 6.6: Relevant part of module SE

6.5.3 Abstract datatype of a generic structure editor

A generic structure editor is a triple of the form (*Tree*, *Path*, *Language*). The *Tree* is an abstract syntax tree as defined in Section 6.5.2. The *Path* is represented by a list of numbers. For instance, if the path is `[2 3]` then the focus tree is the third child of the second child of the root. The *Language* is represented by a signature as defined in Section 6.5.1. Relevant parts of the module defining the editor datatype are shown in Figure 6.6. The boolean function `rootp` tests whether or not the list of numbers representing the path is empty. If so, the whole tree is in focus, otherwise some subtree is in focus. Other functions, such as `tree`, `path`, and `sig` select parts of the structure editor data structure.

6.5.4 Path manipulations

In Section 6.3 we already introduced the path manipulations `root`, `up`, `down`, `next` and `previous`. We now discuss them in some detail and give a specification of their behavior.

Each manipulation, except `root`, may fail in which case the editor is left unchanged. For example, `previous` fails if the focus tree equals the root

```
module Focus-moves
imports SE
exports
context-free syntax
  root(SE)      -> SE
  previous(SE) -> SE
equations
[1]  root([Tree, [Ints], Sig]) = [Tree, [], Sig]
[2a] Int > 1 = true
     ===============
     previous([Tree, [Ints Int], Sig]) = [Tree, [Ints Int - 1], Sig]
[2b] previous([Tree, [Ints 1], Sig]) = [Tree, [Ints 1], Sig]
[2c] previous([Tree, [], Sig]) = [Tree, [], Sig]
```

Figure 6.7: Relevant part of module Focus-moves

and also if the focus is a node which is the leftmost child of its parent. In the first case, the list of integers representing the path is empty, in the second case the list of integers ends with `1`. If a `previous` command succeeds we change the path by decrementing the last element. In the definition of focus moves, shown in Figure 6.7, we present the definitions of `root` and `previous`, other moves are similar and therefore omitted.

6.6 Definition of structured editing

Given the definition of a structure editor consisting of a tree, a path, and a signature, we are in a position to define structured editing itself. We define legal subtree replacements, placeholders and templates, and list editing commands.

6.6.1 Definition of legal subtree replacements

The user is able to perform a number of operations on the focus subtree. However, all these operations must yield a new editor tree that is well-formed as defined in Section 6.3.2.

Figure 6.8 shows the module defining replacement of the focus subtree. Subtree replacement is defined by the function `replace-focus`. This function uses two auxiliary functions `repl-subtree`, and `repl-child`. The legality of the replacement is checked in the conditions of the equations for `replace-focus`. The conditions also distinguish replacing the whole tree

```
module Focus-replacement
imports SE
exports
context-free syntax
  replace-focus(SE, Tree)                          -> SE
  well-formedp(Tree, Signature)                    -> BOOL
  repl-subtr(Tree, Path, Tree)                     -> Tree
  repl-child(Tree, INT, Tree)                      -> Tree
  compatible(Sort, Sort, Signature)                -> BOOL
equations
[1a] well-formedp(Tree, sig(SE)) and
     rootp(SE) = true
     ================
     replace-focus(SE, Tree) = [Tree, path(SE), sig(SE)]
[1b] well-formedp(Tree, sig(SE)) and
     not rootp(SE) and
     compatible (sort(name-to-func(name(Tree), sig(SE))),
                 focus-sort(SE), sig(SE)) = true,
     Tree' = repl-subtr(tree(SE), path(SE), Tree)
     ===========================================
     replace-focus(SE, Tree) = [Tree', path(SE), sig(SE)]
[2a] repl-subtr(Tree, [], Tree') = Tree'
[2b] repl-subtr(Tree, [Int Ints], Tree') =
        repl-child(Tree, Int, repl-subtr(child(Tree, Int), [Ints], Tree'))
[3]  children(Tree) = Trees Tree'' Trees',
     rank(Tree'') = Int,
     Tree''' = [kind(Tree'), name(Tree'), Int; children(Tree')]
     =========================================================
     repl-child(Tree, Int, Tree') =
        [kind(Tree), name(Tree), rank(Tree); Trees Tree''' Trees']
```

Figure 6.8: Relevant part of module Focus-replacement

or only a part. For checking legality, we use the functions `well-formedp`, `compatible`, and `focus-sort`. The first two functions implement the well-formedness and compatibility checks as defined in Section 6.3.2. The `focus-sort` function yields the argument sort of the function corresponding to the parent of the focus. We omitted the equations for other functions and cases where `well-formedp` yields false.

6.6.2 Definition of the placeholder/template mechanism

Until now, we have defined a structure editor which only allows focus moves and subtree replacements. The placeholder/template mechanism, introduced in Section 6.3.2, is defined here.

The module for the placeholder and template based manipulation of trees is shown in Figure 6.9. The focus is either a placeholder or not. In the non-placeholder case we replace the focus tree by a placeholder node corresponding to the argument sort of the parent of the focus. If there is no parent, the sort of the function at the root is used. This is implemented by the function `replace-tree`. In the placeholder case, we compute the set of all functions of compatible sort and we let the user select one of these[2]. Given this function, we replace the placeholder by a template. This is implemented by the function `replace-placeh`. The selected function is mapped to a tree using `make-tree`. If the function to be mapped is a list-constructor function we build a special tree to solve the problem of replacing list placeholders (cf. Section 6.4). Finally, the focus subtree is replaced by the result of `make-tree`. We omitted the equations for other functions and all cases where the `replace-tree` of `replace-placeh` fails as before.

6.6.3 Definition of list editing commands

Commands for editing lists, such as inserting and deleting items were introduced in Section 6.4.3 and are formalized here. A relevant part of a module implementing these commands is shown in Figure 6.10. Recall from Section 6.4.3 that we allow inserting before the focus when the focus is either a list or a list item. In list case, a new item is inserted before all other items in the list and the focus is moved to the newly inserted item (equation `[1a]`). We use an auxiliary function `insert` that creates a new list item node, adds one to the rank of all other items and inserts the new item. The definition of `insert` is omitted. In the list item case, we need to replace the *parent* of the tree in focus. This is defined by replacing the focus tree after applying the function `up` (equation `[1b]`). After inserting, the focus must be moved to the new item, which is defined by copying the path *before* `up` was applied. We omitted equations where the `insert-before` command fails and omitted other commands as well.

[2] How the user selects a function is described in Section 6.7.1 where we connect functionality to the user-interface.

```
module Tree-creation
imports Focus-replacement
exports
context-free syntax
  replace-tree(SE)                      -> SE
  replace-placeh(SE, Function)          -> SE
  make-tree(Function, INT, Signature)   -> Tree
  kind(Tree)                            -> Kind
equations
[1a] kind(focus-tree(SE)) != placeh, path(SE) = [Ints Int],
     Func = sort-to-placeh-func(arg-sort(parent-func(SE), Int), sig(SE))
     ===================================================================
     replace-tree(SE) = replace-focus(SE, make-tree(Func, Int, sig(SE)))
[1b] kind(focus-tree(SE)) != placeh, path(SE) = [],
     Func = sort-to-placeh-func(sort(focus-func(SE)), sig(SE))
     =========================================================
     replace-tree(SE) = replace-focus(SE, make-tree(Func, 0, sig(SE)) )
[2a] kind(focus-tree(SE)) = placeh, path(SE) = [Ints Int]
     =========================================================
     replace-placeh(SE, Func) =
        replace-focus(SE, make-tree(Func, Int, sig(SE)))
[2b] kind(focus-tree(SE)) = placeh, path(SE) = []
     ==================================================
     replace-placeh(SE, Func) =
        replace-focus(SE, make-tree(Func, 0, sig(SE)))
```

Figure 6.9: Relevant part of module Tree-creation

```
module List-editing
imports Tree-creation Focus-moves Focus-replacement
exports
context-free syntax
  insert-before(SE)         -> SE
  focus-is-list(SE)         -> BOOL
  focus-is-list-item(SE)    -> BOOL
equations
[1a] focus-is-list(SE) = true,
     Tree = focus-tree(SE),
     SE' = replace-focus(SE, insert(Tree, 1, sig(SE)))
     ======================
     insert-before(SE) = down(SE')
[1b] focus-is-list-item(SE) = true,
     Tree = focus-parent(SE),
     Int = rank(focus-tree(SE)),
     SE' = replace-focus(up(SE), insert(Tree, Int, sig(SE)))
     ======================
     insert-before(SE) = [tree(SE'), path(SE), sig(SE)]
```

Figure 6.10: Relevant part of module List-editing

6.7 Connection with the user-interface

What have we achieved so far? We have defined:

- the notions *signature* and *term*;
- subtree replacement;
- structured editing based on placeholders and templates; and
- list editing commands.

The next (and final) step is to connect this functionality to the user-interface. The specified structure editor can then be used in a more realistic fashion. The connection with the user-interface consists of:

- writing a SEAL script defining buttons for all editor commands; and
- writing additional ASF+SDF specifications as required by the script.

6.7.1 The SEAL script

Given the algebraic specification of a structure editor, one may generate a user-interface for it by writing a SEAL script for the module SE. A part

```
Configuration for language SE is

button Up
when not SE : rootp(root) enable
  root := Focus-moves : up(root)
doc: "Simulate moving the focus to its parent"

button Down
when Focus-moves : down-enabling(root) enable
  root := Focus-moves : down(root)
doc: "Simulate moving the focus to its first child"

button Template
when Commands : expandable(root) enable
  create("AllFunctions", Commands : make-selector(root));
  Function := select("AllFunctions", Function);
  root := Commands : replace-placeh(root, Function)
doc: "Simulate replacing placeholder by template"

button Placeholder
when not Commands : expandable(root) enable
  root := Commands : replace-tree(root)
doc: "Simulate replacing template by placeholder"
```

Figure 6.11: Part of the SEAL script for terms over module SE

of this script is shown in Figure 6.11. After compiling the script, syntax-directed editors using the language defined by module SE are extended with the user-interface objects defined in the script. Using this mechanism leads to a situation where we have an editor E, operating on a term over SE, which is extended with buttons corresponding to editor commands defined above. These buttons may be enabled or disabled depending on a *condition* which is also specified in the SEAL script.

Consider, for instance, a button labeled "Up" which simulates the `up` command of the specified editor (a term over SE) by calling the `up` function defined in module Focus-moves (cf. Section 6.5.4). This function takes a term over SE as argument and yields an updated term over SE. The result is then used to replace the SE term in E as specified in the SEAL script. The "Up" button is disabled when the focus of our simulated structure editor is positioned at the root. This can be determined by using the `rootp` function defined in module SE. Likewise, we define buttons for the other commands that move the focus. For some of these, however, a function defining the enabling condition of the button lacks. These are defined in a separate module Commands, shown in Figure 6.12, or at a logical place: the module where the command was defined. We illustrate both cases with an example.

As a first example of defining additional functions, consider the `down` command, defined by the `down` function in module Focus-moves (cf. Section 6.5.4). Recall that each focus moving command may fail and if so, `down` returns its argument unchanged. An enabling condition for the `Down` button is thus easily defined by inspecting if `down` returns its argument changed or unchanged. This is defined by adding a function `down-enabling` to module Focus-moves with the following equations:

```
[6a] SE != down(SE)
     ===============
     down-enabling(SE) = true
[6b] SE = down(SE)
     ===============
     down-enabling(SE) = false
```

Given the functions `down` and `down-enabling` the definition of the `Down` button is straightforward (cf. Figure 6.11).

As a second example of defining auxiliary functions, consider the replacement of a placeholder by a template. This editor command is simulated by the `Template` button. It calls the function `replace-placeh` defined in module Tree-creation (cf. Figure 6.9). This function requires two arguments: a term over SE and a term representing a function in the signature. The latter should be the result of selecting a function from the signature by the user (cf. Section 6.6.2). This can be implemented in the following way. When the `Template` button is pressed, SEAL calls the `make-selector` function in module `Commands`. This function computes all functions in the signature that are applicable, i.e., those functions having a compatible sort. These functions are then used by SEAL in a separate editor instance `AllFunctions` and by using SEAL's `select` statement we obtain the selected function. Next, SEAL calls the `replace-placeh` function providing it with the necessary arguments.

```
module Commands
imports List-editing
exports
  sorts User-selection
context-free syntax
  expandable(SE)                                        -> BOOL
  expandable-functions(SE)                              -> Functions
  funcs-of-sort-or-smaller(Sort, Functions, Signature)  -> Functions
  make-selector(SE)                                     -> User-selection
  "Please select one of these functions:" Functions     -> User-selection
equations
[1a] kind(focus-tree(SE)) = placeh
     =====
     expandable(SE) = true
[1b] kind(focus-tree(SE)) != placeh
     =====
     expandable(SE) = false

[2a] expandable(SE) = true,
     rootp(SE) = false,
     Sig = sig(SE),
     Funcs = funcs-of-sort-or-smaller(focus-sort(SE), functions(Sig), Sig),
     Func = focus-func(SE),
     Funcs = Funcs' Func Funcs''   %%remove the focus function
     =====
     expandable-functions(SE) = Funcs' Funcs''
[2b] expandable(SE) = true,
     rootp(SE) = true
     =====
     expandable-functions(SE) = functions(sig(SE))

[3] make-selector(SE) =
      Please select one of these functions: expandable-functions(SE)
```

Figure 6.12: Relevant part of module Commands

Part	total	SDF	ASF	LL	SEAL
Shared (with SEAL)	232	67	165	-	-
Rest	829	219	610	-	-
Script	87	-	-	-	87
Generated (by SEAL)	569	-	-	569	-
TOTAL	1717	286	775	569	-

Table 6.1: Overview of sizes of source code involved in our structure editor.

6.7.2 Quantification of code involved

How much code was involved to implement our structure editor? To answer this question we use the same counting scheme as before (cf. Section 3.7). The implementation of the editor can be subdivided into two parts: its ASF+SDF specification, and its user-interface. The former part shares code with the SEAL implementation (cf. Section 4.5.5). The latter part was generated by the SEAL compiler. The sizes of the sources involved are listed in Table 6.1.

6.7.3 A user session

After initializing our structure editor with a signature describing a language for a list of Boolean expressions we obtain a situation shown in Figure 6.13. Only the button `Template` is enabled —indicated by a **bold** label—. When this button is pushed, the user is asked to make a choice out of *all* functions in the signature. These choices are displayed in a separate term-editor, shown in Figure 6.14. After selecting `And : Bool Bool -> Bool`, the structure editor is updated and the result is shown in Figure 6.15. After pushing the `Down` button of the editor shown in Figure 6.15, the focus is moved to the first child of the root, which is indicated by the path `[1]`. The result is shown in Figure 6.16. Pushing the `Template` button of the editor shown in Figure 6.16 results in asking the user to make a choice out of functions of the sort `Bool` *only*. These choices are displayed in a separate term-editor again, shown in Figure 6.17. The result of selecting `False : -> Bool` is shown in Figure 6.18.

The user session presented above shows that our specified editor can be used to develop a program in a structural manner. An interesting aspect is the interpretation of the language definition during editing which, in principle, allows dynamically changing it. We will discuss this feature in the next section.

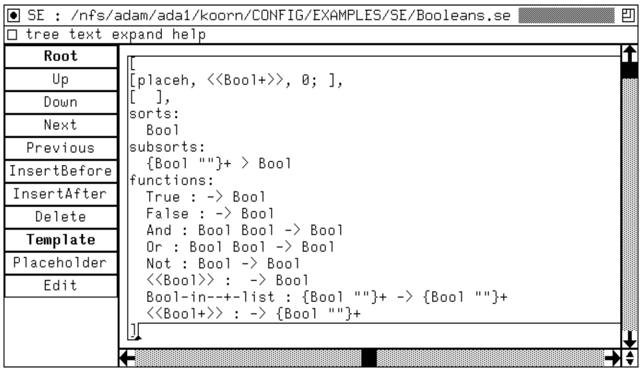


Figure 6.13: The term-editor displaying an initialized structure editor for a list of Boolean expressions.

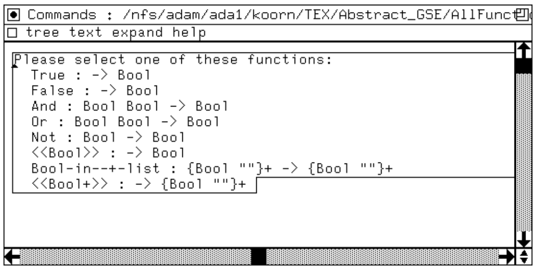


Figure 6.14: All functions in the signature.

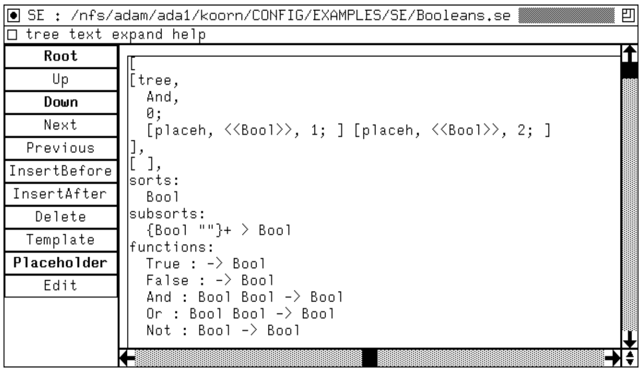


Figure 6.15: The term-editor after replacing the `<<Bool+>>` placeholder by a template corresponding to `And : Bool Bool -> Bool`.

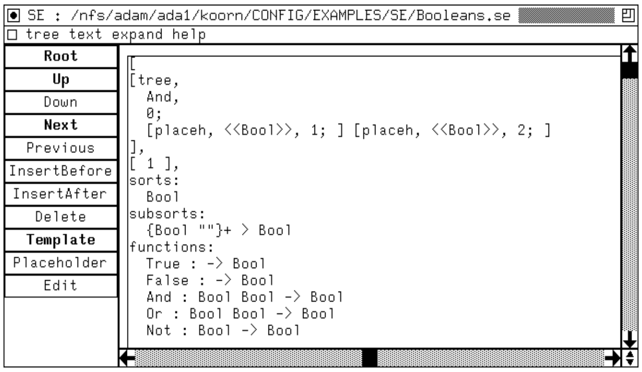


Figure 6.16: The term-editor after pressing the `Down` button.

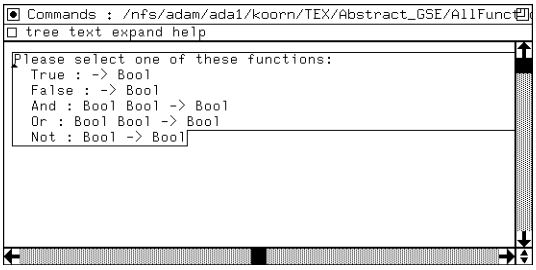


Figure 6.17: All functions of sort `Bool`.

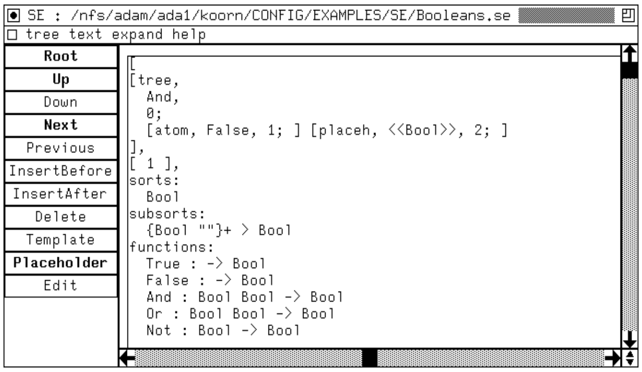


Figure 6.18: The term-editor after selecting `False :  -> Bool`.

6.8 Related work

We will discuss three approaches more or less similar to ours: the work of Schulte [Sch87], based on work of Bertling et al. [BG86], that of Claßen et al. [CL90], and that of Minör [Min90]. A common aspect in the work of these authors and ours is the *interpretation* of the language definition. Most other editor generating systems use the language definition only at generation time, that is, definitions are fed into, for instance, a parser generator. Thus, the generation *result* is used during editing.

Schulte's work amounts to describing a structure editor algebraically and then using the description to derive an implementation in an imperative programming language. The derivation process is hand-crafted as well as the user-interface and extensions of the editor, such as parsing and textual manipulation of programs. The algebraic specification is thus used as a design tool only. Language definitions are based on a combination of a signature and a context-free grammar where there is a one to one correspondence of a function in the signature and a production rule in the grammar. Signatures do not contain subsorts or list-constructor functions, lists are represented as binary trees.

The basis of Schulte's work was [BG86]. There, editing is considered as applying a number of transformations based on conditional rules. For example, a rule might express that a placeholder may be replaced by a template. The most interesting part in this work is the possibility to change the rule set during editing. This allows the user, for instance, to add a rule expressing insertion of a new item in a list. Language definitions are solely based on signatures without subsorts and lists. Lists are represented as binary trees and a default set of transformation rules is derived from the signature. The authors do not present a path mechanism although they mention the availability of an editor prototype. This prototype lacks a graphical user-interface.

Claßen et al. also report the development of a syntax-directed editor based on algebraic specifications. An implementation was derived from the description by transforming it to an OBJ3 [GKK+88] specification which was fed to the OBJ3 system. Here too, editing is considered as applying transformations based on conditional rules. The language definitions lack subsorts and lists are represented as binary trees. The user-interface of the editor is formally specified, but a graphical user-interface is not part of the derived implementation.

Minör's work is less related to ours than the work mentioned above. He does not formally specify a structure editor, but his SbyS editor features interpretation of the language definition used. To our knowledge, SbyS is the only structure editor that uses this technique although it contains generated parts. Language definitions in SbyS are not based on signatures, instead a combination of an abstract syntax definition and a concrete syntax definition is used. There is a one to one correspondence between rules in both definitions. Abstract syntax definitions contain lists, but only in the zero or more variant. A graphical user-interface, based on direct-manipulation [Shn83], is derived automatically from the language definition. In that interface, a user drags a representation of a template from a palette to a representation of a placeholder thus indicating a replacement request. This approach makes the existence of commands to move the focus obsolete.

6.9 Discussion and conclusions

We have formally defined a generic structure editor that uses a signature as language definition. We were able to create an environment which serves as the user-interface of the specified editor by using SEAL. We now discuss advantages, limitations, and suggest issues for future work.

The advantages of our approach are twofold. First, we may study extensions of the editor's command set by adding functions to the ASF+SDF specifications and adapting the SEAL script. The implementations of the ASF+SDF Meta-environment and SEAL allows us to do this fully interactively. Second, by extending the notion of a signature as known in the literature, see e.g. [Wec92], with flat lists our editor behaves more naturally when the focus is moved.

Our approach has three limitations. First of all a *textual view* of the tree built is inevitable. Adding such a view requires the addition of a *keyword skeleton* for each function defined in the signature or extending the language definition with a context-free grammar. The tree part of the editor's data structure must then be *pretty-printed*. This requires an ASF+SDF specification of a language dependent pretty-printer. We consider pretty-printing outside our scope, but the feasibility of it has been shown in [Bra93]. Second, the editor has a fixed set of commands. Although these may be added by adapting the specification, it is still less flexible compared to the transformational approach in the style of Bertling [BG86]. However, we claim that by using SEAL's *active tool* feature, a general transformational style can be achieved. Third, the performance of our editor is such that it can not yet be used for practical program development. This may be improved drastically by compiling the ASF+SDF specification to C using techniques described in [KW93] and by adapting the interface between ASF+SDF and SEAL. For instance, we know that a command to move the focus only changes the focus part of our data structure. In the standard ASF+SDF term-editor we may thus replace only that part instead of replacing the entire data structure which is currently done.

After adding a textual view, the next step is allowing the user to change the textual representation and *parsing* the text fragments that are changed. This requires an algebraic specification of a system generating a parser. Whether or not this is a good idea remains to be investigated.

Chapter 7

Assessment and conclusions

We have presented GSE, a hybrid syntax-directed editor consisting of three parts: text editor, structure editor, and graphical user-interface. It is used in the ASF+SDF Meta-environment of which the user-interface is based on a collection of GSE instances. Furthermore, we have presented SEAL, a user-interface definition language dedicated to the ASF+SDF Meta-environment. Generated tools, such as type-checkers and evaluators, can be connected to the user-interface using SEAL. We assess achievements and state conclusions.

7.1 Assessment

We assess our achievements by discussing to which extent our goals are met. These goals, formulated in Chapter 1, were:

- ensuring the uniformity of all user-interface aspects;
- building an editor which can be used as a generic building block;
- incorporating an existing text editor to obtain first class text editing;
- introducing a mechanism to connect tools to the user-interface; and
- investigating the possibility to generate the editor itself.

Additional goals were: an efficient and easily maintainable implementation, and extensibility and customizability of all editing facilities.

7.1.1 Ensuring uniformity of all user-interface aspects

Our primary goal was to obtain uniformity of all user-interface aspects of the ASF+SDF Meta-environment. The user-interface of this system primarily consists of a collection of editor instances. This goal therefore amounts to ensuring the uniformity of the user-interface aspects of these instances. We have achieved this to a large extent by adopting an approach based on using *generic* software components and *generating* the software needed for their specific use.

7.1.2 Building a generic editor

We have fully achieved our second goal of building an editor which can be used as a generic building block. The key idea here was to parameterize all language dependent parts, such as the parser and the pretty-printer. This led to a situation in which the functionality and behavior of the editor is independent of the language used.

7.1.3 Obtaining first class text editing

Incorporation of an existing text editor, by which first class text editing can be obtained is fully achieved as well. We have incorporated Epoch, an extended version of Emacs, into our editor prototype.

7.1.4 Connecting tools to the user-interface

SEAL provides the mechanism necessary to connect tools to the user-interface of one or more editors. Although we are not completely satisfied with SEAL's current capabilities (cf. Chapter 5), it is our belief that we have made a first step in the right direction.

7.1.5 Generating the editor itself

Our final goal, generating *all* software parts of GSE that are specific for the ASF+SDF Meta-environment, is only partly achieved. We have presented a specification of structure editing facilities and we have shown that the specified editor can be simulated. The reason for only partly achieving this goal is that it is not yet clear to us how to incorporate and use such a specified editor in the ASF+SDF Meta-environment. At this point additional research is required.

7.1.6 Additional goals

Our secondary goals were: an efficient and easily maintainable implementation, and extensibility and customizability of all editing facilities. We briefly discuss these aspects for each part of our software.

Text editing

The efficiency of text editing in GSE is slightly less than in Epoch due to the updating of zone information (cf. Chapter 3).

The way in which we incorporated Epoch, through network communication, has two consequences for maintaining the text editing component. First, although Epoch is maintained by others we now have to maintain the code used for communication and interfacing. Second, the obligation to use the network interface has led to a modular set-up increasing maintainability.

We inherit the extensibility and customizability of Epoch.

Structure editing

Structure editing facilities fall into two groups: built-in facilities —replacing placeholders by templates— which are hard wired in the current code of GSE, and additional facilities obtained by using SEAL.

The efficiency of the facilities in either group is not critical since mouse driven interaction is involved which is intrinsically slow.

Maintenance of built-in facilities is a reason for concern since these are hard wired in the current code. In the future maintainability may be gained when built-in facilities are generated as well (cf. Chapter 6). Additional facilities are easily maintained since both the SEAL script as well as the possibly necessary additional ASF+SDF specification are easily adapted or changed.

Extensibility of structure editing facilities is obtained by using SEAL.

Customizability of built-in facilities is currently impossible, but this may be obtained in the future when they are generated. Additional facilities are customizable by adapting the SEAL script and/or adapting the possibly necessary additional ASF+SDF specification.

User-interface

Efficiency of the user-interface is not critical since pressing a button involves user interaction which is intrinsically slow. Enabling and disabling such buttons by the SEAL run-time system on the other hand is time-critical. Two parts of code are involved here: the code generated by the SEAL compiler and the code used in the SEAL run-time system. Performance of both parts is reasonable, but can be improved.

Maintainability of the user-interface is very good when SEAL is used to generate it. The code used by the SEAL run-time system, the library of graphical objects, and its interface are written by hand and their maintenance is a reason for concern. The SEAL compiler is easily maintained since it is completely specified in ASF+SDF.

The user-interface is partly extensible, again by using SEAL. Extending a user-interface with objects not offered by SEAL requires an extension of the SEAL language, its compiler, and its run-time system.

Customizability at the graphical level —fonts, colors, geometry, etc.— is fully achieved. Epoch allows any combination of font, foreground color, and background color to be used for text. Window colors, fonts used in labels, default sizes of windows, etc. are customizable as well due to the use of OSF/Motif.

7.2 Conclusions

We have built a state-of-the-art hybrid syntax-directed editor. A textual approach was adopted: Epoch, an existing text editor, is used for displaying the result of *all* editing operations. The incorporation of Epoch not only led to rich text editing facilities, but also unified editing inside and outside our system. The user-interface of the editor has a first class graphical appearance and is highly customizable. Furthermore, the user-interface is extensible even at run-time when SEAL is used to generate it. These features make our editor probably the most powerful hybrid syntax-directed editor available to date. SEAL goes beyond generating a user-interface for a single editor, since multiple editor instances may be used to implement a complete user-interface of an interactive programming environment.

Appendix A

SEAL syntax in SDF

```
module SEAL

exports
sorts
  String Module Sort Variable-name Name
  Function-name Dir Dirs Unix-filename
lexical syntax
  "%%" L-Char* "%%"                     -> LAYOUT  %%comment
  [ \t\n]                               -> LAYOUT

  "\\" ~ []                             -> EscChar
  "\\" [01] [0-7] [0-7]                 -> EscChar
  ~ [\000-\037"\\]                      -> L-Char
  EscChar                               -> L-Char
  "\"" L-Char* "\""                     -> String

  [A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]*    -> Module
  [A-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]*    -> Sort
  [a-zA-Z] [a-zA-Z0-9]*                 -> Variable-name
  [a-zA-Z] [a-zA-Z0-9]*                 -> Name
  [a-zA-Z] [A-Za-z0-9\-_]* [A-Za-z0-9]* -> Function-name

  [a-zA-Z0-9_\-]+ "/"                   -> Dir
  "../"                                 -> Dir
  "/" Dir*                              -> Dirs
  "./" Dir*                             -> Dirs
  Dir+                                  -> Dirs
  "\"" Dirs [a-zA-Z0-9_\-.]+ "\""       -> Unix-filename
  "\"" [a-zA-Z0-9_\-.]+ "\""            -> Unix-filename
```

```
sorts
  SEAL-spec SEAL-spec-part Button Menu-entry Menu Entry Active-tool
  Cond-action-pair Tool-CA-pair Docu Tool-Docu Cond-part Action
  Focus-cond-expr Pattern While-cond Focus Focus-expr Focus-action Term
context-free syntax
  "Configuration" for language Module is SEAL-spec-part+ -> SEAL-spec
  Button                                                 -> SEAL-spec-part
  Menu-entry                                             -> SEAL-spec-part
  Menu                                                   -> SEAL-spec-part
  Active-tool                                            -> SEAL-spec-part

  button Name Cond-action-pair+ Docu              -> Button
  start-stop button Name Cond-action-pair+ Docu   -> Button
  menu entry Name in Name Cond-action-pair+ Docu  -> Menu-entry
  menu Name ":" Entry+                            -> Menu
  Name "," Cond-action-pair+ Docu                 -> Entry
  active tool Name Tool-CA-pair+ Tool-Docu        -> Active-tool

  when Cond-part enable {Action ";"}+             -> Cond-action-pair
  enable {Action ";"}+                            -> Cond-action-pair
  when Cond-part do {Action ";"}+                 -> Tool-CA-pair
  do {Action ";"}+                                -> Tool-CA-pair

  Focus-cond-expr is Sort                         -> Cond-part
  Focus-cond-expr matches Pattern                 -> Cond-part
  String                                          -> Pattern
  Focus-cond-expr "is-meta-var"                   -> Cond-part
  Module ":" Function-name({Focus-cond-expr ","}*) -> Cond-part
  not Cond-part                                   -> Cond-part
  "(" Cond-part ")"                               -> Cond-part {bracket}
  Cond-part and Cond-part                         -> Cond-part {left}
  Cond-part or  Cond-part                         -> Cond-part {left}

  focus {Focus-action ","}*                       -> Focus-cond-expr
  Unix-filename "." focus {Focus-action ","}*     -> Focus-cond-expr

  Focus-cond-expr is Sort                         -> While-cond
  Focus-cond-expr matches Pattern                 -> While-cond
  Focus-cond-expr "is-meta-var"                   -> While-cond
  Module ":" Function-name({Variable-name ","}*)  -> While-cond
  focus {Focus-action ","}+                       -> While-cond
  Unix-filename "." focus {Focus-action ","}+     -> While-cond
  Variable-name "." focus {Focus-action ","}+     -> While-cond
  not While-cond                                  -> While-cond
  "(" While-cond ")"                              -> While-cond {bracket}
  While-cond and While-cond                       -> While-cond {left}
  While-cond or  While-cond                       -> While-cond {left}
```

```
  Focus ":=" Term                                    -> Action
  Variable-name ":=" Term                            -> Action
  Variable-name ":=" Focus {Focus-action ","}*       -> Action
  Focus-expr                                         -> Action
  while While-cond do {Action ";"}+ od               -> Action
  Module ":" Function-name({Variable-name ","}*)     -> Action
  create(Unix-filename, Module, Text, Sort)          -> Action
  create(Variable-name, Module, Text, Sort)          -> Action
  create(Unix-filename, Variable-name)               -> Action
  create(Variable-name, Variable-name)               -> Action
  create(Unix-filename, Term)                        -> Action
  create(Variable-name, Term)                        -> Action

  Focus {Focus-action ","}*                          -> Focus-expr
  {Focus-action ","}+                                -> Focus-expr

  focus                                              -> Focus
  Unix-filename "." focus                            -> Focus
  Variable-name "." focus                            -> Focus

  up                                                 -> Focus-action
  down                                               -> Focus-action
  previous                                           -> Focus-action
  next                                               -> Focus-action
  root                                               -> Focus-action
  save                                               -> Focus-action
  restore                                            -> Focus-action

  Variable-name                                      -> Term
  select "(" Unix-filename "," Sort ")"              -> Term
  select "(" Variable-name "," Sort ")"              -> Term
  Module ":" Function-name({Variable-name ","}*)     -> Term

  String+                                            -> Text
  readfile "(" Unix-filename ")"                     -> Text
  readfile "(" Variable-name ")"                     -> Text

  doc ":" String manual entry ":" Name               -> Docu
  doc ":" String                                     -> Docu
  manual entry ":" Name                              -> Tool-Docu

priorities not Cond-part -> Cond-part >
           Cond-part and Cond-part -> Cond-part >
           Cond-part or Cond-part -> Cond-part
priorities not While-cond -> While-cond >
           While-cond and While-cond -> While-cond >
           While-cond or While-cond -> While-cond
```

Bibliography

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bar84] H.P. Barendregt. *The Lambda Calculus; its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathatematics.* North-Holland, 1984.

[BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. Rapports de Recherche 327, INRIA, Sophia-Antipolis, 1984.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SIGSOFT'88)*, Boston, 1988.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 24(2).

[BCG86] G. Berry, P. Couronne, and G. Gonthier. Synchronous programming of reactive systems: An introduction to ESTEREL. In *Proceedings of the first France-Japan Symposium on Artificial Intelligence and Computer Science, Tokyo.* North Holland, 1986. Also appeared as INRIA Rapport de Recherche No. 647.

[Ber92] T. Berlage. Using taps to separate the user interface form the application code. In *Proceedings of UIST '92*, pages 191–198, Monterey, CA, November 1992.

[BG86] H. Bertling and H Ganzinger. A Structure Editor Based on Term Rewriting. In The Commission of the European Communities, editor, *Esprit '85 - Status Report of Continuing Work 1*, pages 455–466. Elsevier, 1986.

[BGV92] R.A. Ballance, S.L. Graham, and M.L. Van De Vanter. The Pan Language-Based Editing System. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, 1992.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[BL90] J. van den Bos and C. Laffra. Project DIGIS Building Interactive Applications by Direct Manipulation. *Computer Graphics Forum*, 9(3):181–193, september 1990.

[Bos88] J. van den Bos. Abstract interaction tools: A language for user interface management systems. *ACM Transactions on Programming Languages and Systems*, 10(2):215–247, 1988.

[Bra92] M.G.J. van den Brand. *Pregmatic, A generator for incremental programming environments.* PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[Bra93] M.G.J. van den Brand. Generation of language independent modular prettyprinters. Report P9327, Programming Research Group, University of Amsterdam, 1993.

[BS86a] R. Bahlke and G. Snelting. Context-sensitive editing with PSG environments. In R. Conradi, T.M. Didriksen, and D.H. Wanvik, editors, *Proceedings of the International Workshop on Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 26–38. Springer-Verlag, 1986.

[BS86b] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[BS92] R. Bahlke and G. Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467–502, October 1992.

[Che87] S. Chernicoff. *Programming with the Toolbox*, volume 2 of *Macintosh Revealed.* Hayden Books, 4300 West 62nd Street, Indianapolis, Indiana 46268, USA, second edition, 1987.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[CI88] D. Clément and J. Incerpi. Specifying the behavior of graphical objects using ESTEREL. Rapports de Recherche 836, INRIA, Sophia Antipolis, 1988.

[CL90] I. Claßen and M. Löwe. Algebraic development of a syntax directed editor. Technivcal report 90/37, Technical University of Berlin, Berlin, 1990.

[CP85] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. In *Conference proceedings of SIGGRAPH '85*, pages 199–204. ACM, 1985. Appeared as *Computer Graphics* 19(3).

[Des88] Th. Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[Deu92] A. van Deursen. Specification and generation of a λ-calculus environment. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 14–26. SION, 1992. Also appeared as Report CS-R9233, Centrum voor Wiskunde en Informatica (CWI) Amsterdam.

[DGHKL80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the MENTOR experience. Rapports de Recherche 26, INRIA, Rocquencourt, 1980.

[DGHKL84] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the MENTOR experience. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 128–140, New York, 1984. McGraw-Hill.

[DK89] M.H.H. van Dijk and J.W.C. Koorn. *Generic syntax editor.* INRIA, Sophia-Antipolis, 1989. In: The CENTAUR Documentation - Version 0.9, Volume I - User's Guide.

[DK90] M.H.H. van Dijk and J.W.C. Koorn. GSE, a generic syntax-directed editor. Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1990.

[DKT93] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *Journal of Symbolic Computation*, 15:523–545, 1993. Special Issue on Automatic Programming.

[DS90] P. Dewan and M. Solomon. An Approach to Support Automatic Generation of User Interfaces. *ACM Transactions on Programming Languages and Systems*, 12(4):566–609, 1990.

[Eck80] R. Eckert. Specification of Graphics Systems. In Guedj et al, editor, *IFIP-WG 5.2 Workshop on the Methodology of Interaction*, pages 195–209. North Holland, 1980.

[Epo92] University of Illinois, Urbana-Champaign. *Epoch, GNU Emacs for the X Windowing System*, 1992. Release 4.0, based on GNU Emacs 18.58.

[Fou90] Open Software Foundation. *OSF/Motif Programmer's Guide, Revision 1.1.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1990.

[GF92] D.F. Gieskens and J.D. Foley. Controlling user interface objects through pre- and postconditions. In *Conference proceedings of CHI '92*, pages 189–194. Addison Wesley, 1992.

[GKK+88] J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.

[Gor88] M.J.C. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[GP91] J.F. Groote and A. Ponse. μCRL: A base for analysing processes with data. In E. Best and G. Rozenberg, editors, *Proceedings of the third Workshop on Concurrency and Compositionality*, volume 191 of *GMD-Studien*, pages 125–130, 1991.

[Han71] W.J. Hansen. User engineering principles for interactive systems. In *Proceedings of the AFIPS Conference*, volume 39, pages 523–532, Reston, Virginia, 1971. AFIPS Press.

[Hee92] F.C. Heeman. State-of-the-Art Window Systems and UIMSs. Technical Report 92-07, Software Engineering Research Centrum, Utrecht, the Netherlands, march 1992.

[HH89] H.R. Hartson and D. Hix. Human-computer interface development: Concepts and systems for its management. *ACM Computing Surveys*, 21(1), 1989.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hil86] R.D. Hill. Supporting concurrency, and synchronization in human-computer interaction — the sassafras UIMS. *ACM Transactions on Programming Languages and Systems*, 5(3):179–210, 1986.

[HK89] J. Heering and P. Klint. The syntax definition formalism SDF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 283–297. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 6.

[HKKL86] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of Interactive Programming Environments. In The Commission of the European Communities, editor, *Esprit '85 - Status Report of Continuing Work 1*, pages 467–477. Elsevier, 1986.

[HN86] A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[ISO87] ISO. *Information processing systems — open systems interconnection — LOTOS — a formal description technique based on the temporal ordering of observational behaviour*, 1987. ISO/TC97/SC21/N DIS8807.

[Jac86] R.J.K. Jacob. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics*, 5(4):283–317, 1986.

[JMB+93] I. Jacobs, F. Montagnac, J. Bertot, D. Clément, and V. Prunet. The Sophtalk Reference Manual. Rapports de Recherche 150, INRIA, Sophia Antipolis, February 1993.

[Kah87] G. Kahn. Natural semantics. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[KB93] J.W.C. Koorn and H.C.N. Bakker. Building an editor from existing components: an exercise in software re-use. Report P9312, Programming Research Group, University of Amsterdam, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as KB93.ps.Z.

[KJT+93] H. Korte, H. Joosten, V.G. Tijssen, A. Wammes, and J. Wester. Realization of a LOTOS simulator with ASF+SDF, 1993. In: ESPRIT Project 2177, Generation of Interactive Programming Environments II, Fifth annual review report.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Dietz, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'92*, pages 168–177. SION, 1992. Appeared as Report P9202, University of Amsterdam. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92b.ps.Z.

[Koo93] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. In H.A. Wijshoff, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'93*, pages 217–228. SION, 1993. Also appeared as Report P9222, University of Amsterdam. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Koo92a.ps.Z.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. To appear in Conference Proceedings of Computing Science in the Netherlands, CSN'93. Also available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[Lan85] B. Lang. Mentor - design and implementation of the kernel of a program manipulation system. In J. McDermid, editor, *Integrated project support environments*, volume 1, pages 175–188. IEEE Software Engineering Series, 1985.

[Lan86] B. Lang. On the usefulness of syntax directed editors. In R. Conradi, T.M. Didriksen, and D. Wanvik, editors, *Advanced Programming Environments*, volume 244 of *Lecture Notes in Computer Science*, pages 47–51. Springer-Verlag, 1986.

[LeL91] INRIA, Rocquencourt. *LeLisp, Version 15.24, Reference Manual*, 1991.

[Ler92] B. S. Lerner. Automated customization of structure editors. *International Journal of Man-Machine Studies*, 37(4):529–563, October 1992.

[LLG90] D. Lewis, D. LaLiberte, and GNU Manual Group. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Inc., 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 1.03 edition, December 1990. Describes Emacs Version 18.

[Log88] M.H. Logger. An integrated text and syntax-directed editor. Report CS-R8820, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988.

[MBD+90] B. Magnusson, M. Begtsson, L.O Dahlin, G. Fries, A. Gustavson, G. Hedin, S. Minör, D. Oscarsson, and M. Taube. An overview of the mjølner/orm environment: Incremental language and software development. Technical Report LU-CS-TR:90-57 and LUTEDX/(TECS-3026)/1-12/(1990), Lund University and Lund Institute of Technology, 1990.

[MGD+90] B.A. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet — comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11), 1990.

[Min90] S. Minör. *On Structure-Oriented Editing.* PhD thesis, Lund University, 1990.

[Mor81] T. P. Moran. The Command Language Grammar: a representation for the user interface of interactive computer systems. *International Journal of Man-Machine Studies*, 15:3–50, 1981.

[MSK90] N.P. Mayer, A. W. Shepherd, and A. J. Kuchinsky. Winterp: an object-oriented, rapid prototyping, development and delivery environment for building extensible applications with the OSF/Motif UI toolkit. In *Proceedings of Xhibition '90*, San Jose, 1990.

[MV90] S. Mauw and G.J. Veltink. A process specification formalism. *Fundamenta Informaticae*, XIII:85–139, 1990.

[Mye90] B.A. Myers. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems*, 12(2):143–177, 1990.

[NHE+83] D. Notkin, N. Habermann, R. Ellison, G. Kaiser, and D. Garlan. Response to Waters' article on structure oriented editors. *SIGPLAN Notices*, 18(4), 1983. Correspondence section.

[Not85] D. Notkin. The GANDALF project. *The Journal of Systems and Software*, 5(2):91–105, 1985.

[NS90] L. Neal and G. Szwillus. Report on the CHI '90 Workshop on Structure Editors. *SIGCHI bulletin*, 22(2):49–53, 1990.

[Ols87] D.R. Olsen. ACM SIGGRAPH Workshop on Software Tools for User Interface Management. *Computer Graphics*, 21(2):71–147, 1987.

[Par72] D. L. Parnas. A Technique for Software Module Specification with Examples. *Communications of the ACM*, 15(5):330–336, 1972.

[Par90] H. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development.* Springer-Verlag, 1990.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Rek92.ps.Z.

[Rem92] B. Remington. CHIRP: The Computer-Human Interface Rapid Prototyping Toolkit. In *Conference proceedings of CHI '92*, pages 233–234. Addison Wesley, 1992.

[RK91] J. Rekers and W. Koorn. Substring parsing for arbitrary context-free grammars. In *Proceedings of the Second International Workshop on Parsing Technologies, IWPT'91*, pages 218–224. Association for Computational Linguistics, 1991. Also in: *SIGPLAN Notices*, 26(5):59-66,1991.

[RT89a] T. Reps and T. Teitelbaum. *The Synthesizer Generator: a System for Constructing Language-Based Editors.* Springer-Verlag, 1989.

[RT89b] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual - Third edition.* Springer-Verlag, 1989.

[Sch87] W. Schulte. Algebraische spezifikation und programmentwicklung eines syntaxgesteuerten editors. Master's thesis, Technische Universitaet Berlin, 1987. In German.

[SG86] R.W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(3):79–109, 1986.

[Sha83] U. Shani. Should program editors not abandon text oriented commands? *SIGPLAN Notices*, 18(1), 1983.

[Shn83] B. Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer*, 16(8):57–69, 1983.

[SM88] M. R. Szczur and P. Miller. Transportable Applications Environment (TAE) PLUS Experiences in "Object"ively Modernizing a User Interface Environment. In *OOPSLA '88 Proceedings*, pages 58–70, 1988. Appeared as *SIGPLAN Notices*, vol. 23, no. 11, November 1988.

[Sta81] R.M. Stallman. Emacs, the extensible, customizable, self-documenting display editor. In *ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–160, Portland, Ore., 1981. Appeared as *SIGPLAN Notices*, vol. 16, no. 6, June 1981.

[SY88] M. L. Scott and S. Yap. A grammar-based approach to the automatic generation of user-interface dialogues. In *Proceedings of SIGCHI '88, Human Factors in Computing Systems*, pages 73–78, 1988.

[TR81] T. Teitelbaum and T.W. Reps. The Cornell Program Synthesizer: syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[Wat82] R.C. Waters. Program editors should not abandon text oriented commands. *SIGPLAN Notices*, 17(7), 1982.

[Wec92] W. Wechler. *Universal Algebra for Computer Scientists*, volume 25 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992.

[WL] J.J. van Wijk and R. van Liere. "What You Draw Is What You Control" A Toolkit for Computational Steering. to appear.

[WR82] P.C.S. Wong and E.R. Reid. Flair - user interface design tool. *Computer Graphics*, 16(3):87–98, 1982.

Het genereren van uniforme gebruikersinterfaces voor interactieve programmeeromgevingen

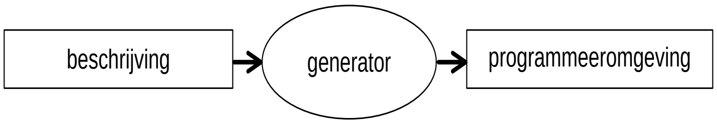
Inleiding

Het genereren van software, in plaats van het met de hand schrijven daarvan, is tegenwoordig een op grote schaal toegepaste methode en het wetenschappelijk onderzoek op dit terrein bloeit. Zo zijn er een aantal onderzoeksprojecten met als doel het genereren van programmeeromgevingen vanuit een formele beschrijving. In een dergelijke beschrijving zijn twee delen te onderscheiden: een definitie van een (programmeer) taal en een beschrijving van het gebruikersinterface.

Het genereren van interactieve programmeeromgevingen

Een *programmeeromgeving* is een verzameling hulpmiddelen die het programmeren vereenvoudigen. Deze verzameling bestaat in het algemeen uit een *editor* (tekstverwerker voor programma's), een *typechecker* die controleert dat bepaalde fouten in het programma niet voorkomen en een *evaluator* die het programma uitvoert. Al deze hulpmiddelen zijn zelf ook weer programma's.

Systemen (programma's) die programmeeromgevingen genereren zullen met behulp van een generator deze hulpmiddelen afleiden/opleveren uit een beschrijving ervan. Het algemene model dat hierbij wordt gebruikt is afgebeeld in Figuur 1. In de hulpmiddelen van een gegenereerde omgeving komen onderdelen voor die onafhankelijk van de beschrijvingen zijn, dat



Figuur 1. Model voor het genereren van programmeeromgevingen

wil zeggen ze zullen in iedere gegenereerde omgeving voorkomen. Deze onderdelen kunnen eveneens door de generator worden gegenereerd, maar het is efficiënter om deze onderdelen generiek te maken. De meeste systemen gebruiken dit principe. Een typisch voorbeeld van een generiek onderdeel is de editor: het gebruikersinterface van de editor en de editing faciliteiten zijn immers onafhankelijk van de beschrijving. Aan het (eenmalig) genereren van de generieke onderdelen zelf wordt echter weinig aandacht besteed. Bij een editor betekent dit dat het (grafische) gebruikersinterface en de editing faciliteiten gegenereerd worden.

De meeste systemen gebruiken *abstracte syntax bomen* [ASU86] als intern data formaat voor programma's. Deze boomrepresentatie wordt "zichtbaar" via de editor, namelijk via de manier waarop programma's gemaakt en gewijzigd kunnen worden. We onderscheiden drie verschillende vormen van editors. De *tekst*editor, zoals Emacs [Sta81], waarbij de gebruiker alleen met tekst werkt. De *structuur*editor, zoals Emily [Han71], waarbij de gebruiker alleen met de boom structuur (in tekstuele vorm natuurlijk) kan werken. De *hybride* editor waarbij de gebruiker zowel de tekst als de boomstructuur kan manipuleren. De meer algemene term *syntax gestuurde* editor staat voor zowel structuur- als hybride editors. Het gebruik van syntax gestuurde editors in gegenereerde programmeeromgevingen heeft als voordeel dat de abstracte syntax boom direct beschikbaar is voor de andere hulpmiddelen in de omgeving.

In dit proefschrift beperken we onszelf tot het gebruikersinterface en alle editing aspecten van de *ASF+SDF Meta-omgeving* [Kli93]. Het unieke van dit systeem is dat de ontwikkelomgeving (waarin men de beschrijving maakt) en de gegenereerde omgeving volledig zijn geïntegreerd. Zowel de beschrijving van een programmeertaal als programma's in die taal zijn tegelijk aanwezig, dit is mogelijk doordat verschillende editors tegelijk actief kunnen zijn. Andere hulpmiddelen in een door dit systeem gegenereerde omgeving worden algebraïsch beschreven en worden geïmplementeerd door middel van termherschrijven. Ieder hulpmiddel gebruikt een abstracte syntax boom —of *term*— als dataformaat voor zowel invoer als uitvoer.

Beschrijven van gebruikersinterfaces

Projecten met als doel het genereren van programmeeromgevingen besteden maar weinig aandacht aan het genereren van generieke onderdelen zelf, zoals het grafische gebruikersinterface. Er zijn echter projecten die zich volledig concentreren op het genereren van grafische gebruikersinterfaces, waarbij de "*gereedschapskist*"- en de "*management systeem*"-methode onderscheiden kunnen worden. We zullen deze nu kort bespreken en bezien welke consequenties deze hebben binnen onze context.

De "gereedschapskist"-methode om een grafisch gebruikersinterface te maken kan worden gekarakteriseerd als een techniek waarbij een software bibliotheek wordt gebruikt. Deze bibliotheek bevat in het algemeen een zeer groot aantal functies. Er zijn functies om grafische objecten, zoals buttons, te maken en functies voor de layout van een scherm.

Er bestaan ook methoden om gebruikersinterfaces in hun geheel te genereren, zogenaamde "management systemen" [HH89] voor gebruikersinterfaces. Deze dienen om gebruikersinterfaces te definieren, ze hebben controle over het gebruikersinterface tijdens executie en ze verbinden het grafische deel met het niet grafische deel van een interactief programma.

In onze context kunnen beide methoden niet worden ingezet zonder het met de hand schrijven van extra software. "Gereedschapskisten" vergen het schrijven van extra software voor de layout van het scherm. "Management systemen" hebben gebrek aan kennis van de overige (niet grafische) software. Dit speelt een rol als *dynamische* wijzigingen in het gebruikersinterface afhangen van datawaarden in de niet grafische delen. Het probleem hier is dat "management systemen" ontworpen zijn voor *algemeen* gebruik, ze zijn niet toegespitst op het genereren van programmeeromgevingen.

Onze doelen

Het verkrijgen van *uniformiteit* in het gebruikersinterface van de ASF+SDF Meta-omgeving is ons hoofddoel. Dit impliceert dat we moeten voorkomen dat gebruikers te maken krijgen met nieuwe editor commando's. Daarnaast heeft het bouwen van een editor voor dit systeem twee andere implicaties. Ten eerste, er worden meerdere instanties van de editor gebruikt, de editor moet dus als een generieke bouwsteen kunnen worden gebruikt. Ten tweede, schrijvers van ASF+SDF specificaties kunnen *willekeurige* hulpmiddelen definieren die werken op abstracte syntax bomen. Deze hulpmiddelen moeten worden "verbonden" met de editor, we hebben dus een uitbreidbaar gebruikersinterface voor de editor nodig en een "verbindingsmechanisme". Secondaire doelen zijn: een efficiënte, makkelijk te onderhouden implementatie en uitbreidbaarheid zowel als aanpasbaarheid van alle editing faciliteiten. Samengevat zijn onze doelen:

- het verzekeren van uniformiteit van alle gebruikersinterface aspecten;
- het bouwen van een editor, te gebruiken als een generieke bouwsteen;
- het integreren van een bestaande teksteditor;
- het introduceren van een verbindingsmechanisme; en
- het onderzoeken van de mogelijkheid om de editor zelf te genereren.

De verschillende hoofdstukken beschrijven hoe we deze doelen (grotendeels) hebben bereikt.

GSE: een generieke syntax-gestuurde editor

Hoofdstuk 2[1] bespreekt een model voor het integreren van tekst- en structuurediting. Dit model is gebruikt om een prototype editor te bouwen waarvan het gebruikersinterface werd gemaakt met behulp van de gfxobj "gereedschapskist" [CI88]. Het gebruik van dit prototype als generieke bouwsteen heeft geleid tot uniformiteit van structuur editing en ook tot uniformiteit van zowel het uiterlijk (grafische delen) als het gedrag (reactie op gebeurtenissen) van het gebruikersinterface.

GSE en Emacs

In Hoofdstuk 3[2] beschrijven we het vervangen van de tekstediting faciliteiten van het prototype door Emacs, een bestaande tekst editor met een zeer groot aantal commando's. Verder wordt het vervangen van het op gfxobj gebaseerde gebruikersinterface door een gebaseerd op OSF/Motif [Fou90] besproken. We hebben daarbij de software voor structuurediting in het prototype hergebruikt en de beide andere delen (tekstediting en gebruikersinterface) vervangen, resulterend in een *gedistribueerde* editor. Het incorporeren van Emacs heeft geleid tot: uniformiteit van tekstediting binnen en buiten de ASF+SDF Meta-omgeving, beter te onderhouden software en tot zowel uitbreidbare als aanpasbare faciliteiten voor tekstediting. Het

[1] Dit hoofdstuk is een revisie van [Koo92]

[2] Dit hoofdstuk is een revisie van [KB93] en is een coproduktie met H.C.N. Bakker

gebruik van OSF/Motif bevorderde de uniformiteit ook: het wordt veel toegepast en dus zijn vele gebruikers reeds bekend met het gedrag.

SEAL: definities van gebruikersinterfaces

Het koppelen van hulpmiddelen aan de editor en het uitbreiden van het gebruikersinterface is het onderwerp van Hoofdstuk 4[3]. We beschrijven SEAL, een op de ASF+SDF Meta-omgeving toegespitste taal om gebruikersinterfaces te beschrijven. Deze taal is dusdanig ontworpen dat hulpmiddelen gemakkelijk te koppelen zijn aan editors. De SEAL vertaler (generator) maakt het met de hand schrijven van extra software overbodig. Dit heeft geleid tot uniformiteit van de uitbreidingen van het gebruikersinterface en maakt tevens structuurediting uitbreidbaar. Bovendien wordt hetzelfde systeem gebruikt voor zowel het schrijven van definities van hulpmiddelen als voor het beschrijven van gebruikersinterfaces, hetgeen uniformiteit bevordert. Verder is de SEAL vertaler in ASF+SDF geschreven en is dus gemakkelijk te wijzigen, uit te breiden en te onderhouden.

Gebruik van SEAL: case-studies

De taal SEAL, geïntroduceerd in Hoofdstuk 4 is breder toepasbaar dan alleen voor het koppelen van hulpmiddelen aan gebruikersinterfaces. Het biedt bijvoorbeeld de mogelijkheid om editors te laten samenwerken. Naast het illustreren van SEAL's potentiële kracht en het geven van een overzicht van het gebruik van SEAL is Hoofdstuk 5 voornamelijk bedoeld om vast te stellen wat het gemak is waarmee men gebruikersinterfaces definieert. Hiervoor presenteren we vijf case-studies: een "klassieke" programmeeromgeving, programmatransformaties, interactieve invoer en uitvoer, het simuleren van parallelle systemen en onderling afhankelijke editors.

Een specificatie van structuur editing

In Hoofdstuk 3 hebben we alle faciliteiten voor tekstediting vervangen door een bestaande tekst editor. Hoofdstukken 4 en 5 beschrijven het genereren van het gebruikersinterface van een editor. De laatste stap in dit proces is het genereren van de derde en laatste component van een editor: de faciliteiten voor structuurediting. Hoofdstuk 6 bespreekt de mogelijkheden

[3]Dit hoofdstuk is een revisie van [Koo93]

om dit te bereiken. We presenteren een formele, algebraïsche, definitie in ASF+SDF van een generieke structuureditor. Deze kan worden gesimuleerd in een gegenereerde omgeving, waarbij we SEAL gebruiken om de commando's van de editor te modelleren. Dit hoofdstuk kan worden beschouwd als een eerste stap in de richting van een door zichzelf gegenereerde variant van de ASF+SDF Meta-omgeving, maar de gepresenteerde definitie kan ook worden gebruikt om te bestuderen wat structuurediting exact is.

Previous titles in the ILLC Dissertation Series:

Transsentential Meditations; Ups and downs in dynamic semantics
Paul Dekker
ILLC Dissertation series, 1993-1

Resource Bounded Reductions
Harry Buhrman
ILLC Dissertation series, 1993-2

Efficient Metamathematics
Rineke Verbrugge
ILLC Dissertation series, 1993-3

Extending Modal Logic
Maarten de Rijke
ILLC Dissertation series, 1993-4

Studied Flexibility
Herman Hendriks
ILLC Dissertation series, 1993-5

Aspects of Algorithms and Complexity
John Tromp
ILLC Dissertation series, 1993-6

The Noble Art of Linear Decorating
Harold Schellinx
ILLC Dissertation series, 1994-1

Generating Uniform User-Interfaces for Interactive Programming Environments
Jan Willem Cornelis Koorn
ILLC Dissertation series, 1994-2