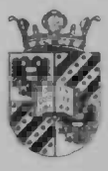
Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep
Informatica

Building TEMPLES on ICE

The replacement of IGLOO by a component generator

Eibert Engelsman



begeleiders:
Dr.ir. H.B.M. Jonkers (Philips Research)
Prof.dr.ir. L.M.G. Feijs (Philips Research)
Prof.dr. G.R. Renardel de Lavalette (RuG)

april 1995

RuG

Abstract

Re-use of software is a key issue in current software engineering processes. The importance of software reuse is indicated by the amount of redundant work saved by using well-documented, well-specified libraries.

A precondition for the flexible use of these libraries is that they are in some way *generic*, i.e. adaptable to current needs of the software engineer.

In this thesis we will describe the development of a language suited for expressing this genericity, and the development of a tool that can *instantiate* a given generic expression to a specific instantiation.

The project has been worked out at the Philips Nat.Lab. in Eindhoven, and has been tailored to be used in the SPRINT method, a software engineering method especially suited for embedded software development, a field in which software re-use can play a major role in reducing development lead times.

Contents

Preface

Today, the world and the complex interactions of the objects dependent on it is hard to imagine without the use of information technology. Since the invention of the first electrical computers there has been a rat race between the increasing possibilities of the hardware and the increasing demands of the software and their users.

For years now, we are in a situation where this increasing complexity must somehow be controlled by the system designers. The size of current systems introduces the need to use methods by which the complexity can be coped with.

In these methods, an important issue is *re-use*. Re-use of existing software parts allows the designers to focus on the new elements of the system instead of re-creating the old parts. Tested parts with a clear interface can then be used without risks, thus reducing development lead times.

We shall see that the SPRINT method of software engineering employed at Philips practically and formally supports software re-use to a large extent.

However, to enlarge the domain of re-usable parts and to reduce the complexity of the SPRINT-tools that analyse re-use, a new method for re-use had to be introduced.

In this thesis, we will discuss the system that implements this method. The method allows a user to describe a parameterized re-usable component in the macro-language TEMPL. This component may be instantiated with the TEMPLES tool by providing the actual parameters needed in a design.

To allow the user a clear overview of the contents of the re-usable library and the use of the parameters in the re-usable components, it is possible to use an easy-to-use graphical user interface.

We shall see that the way we have implemented the language and the instantiation tool allows the use of the system for any text-based development language.

Outline of the thesis

In chapter 1 we will introduce the context of the assignment. The assignment itself is focused on the SPRINT software engineering method, but our system will be able to support many more methods.

Chapter 2 shows how the system was developed. The development was based on a refining of the requirements and in this chapter we will see that this way of development worked out perfectly for the system.

Chapter 3 introduces the macro-language TEMPL by means of an annotated example, its syntactical constructs, its semantics and pragmatics.

In the appendices the reader can find a.o. a description of the system in the form of a user's manual of TEMPLES, a description of the tools used for the implementation of the system and some TEMPL examples.

Acknowledgements

I would like to thank here all the people that made the work for this thesis possible, especially Hans Jonkers, for his suggestful solutions to many of my problems, Loe Feijs, who, apart from useful comments on earlier versions of this thesis, suggested the more general approach to the language design, Gerard Renardel, without whom many of the ideas expressed in this document would be incomprehensible to the reader, Erik Saaman, Tineke de Bunje and Roeland van de Bos, for other useful suggestions and Marije Withaar, for her love, support and patience during my work on this thesis.

Chapter 1

The Assignment and its Context

Each project has a context in which it is embedded. We give this context by introducing the reader in this chapter to the world of software engineering at Philips. We discuss the SPRINT method, which is a software engineering method especially suited for embedded software development. We explain the basic language elements of the COLD language family, because specific aspects of COLD were at the base of the assignment. We then describe the assignment, and hope that the given introduction clarifies the assignment.

1.1 Philips and Software

Although it may seem that Philips as a major electronics company should have little interest in software as opposed to hardware, Philips has realized that future products need to be *flexible*, e.g. easily extendible with new functionality, *reusable*, in the sense that parts of one product may be used in another product and that products should be developed with a *short development time*. Furthermore, the competition in the consumer electronics market requires the introduction of ever more new features, implying increasing levels of complexity.

The answer to these issues is embedded software, which is software that is designed to operate closely with the hardware that it controls. Software engineering introduces its own specific processes and problems. For example, the amount of software in a television set doubles approximately every two years. The sector IST (Information and Software Technology), which is located at the Nat.Lab. in Eindhoven supports the software engineering process with a wide range of research activities. The IST has, among other things, developed and currently supports the SPRINT method for software engineering, which is especially suited for embedded software systems.

1.2 The SPRINT method

The SPRINT method is a formal method for the development of control software of audio/video (A/V) systems. SPRINT is an acronym for Specification, Prototyping and Reusability INTegration. We give an overview of the concepts of SPRINT that are essential to this project, for a more thorough overview of the method we refer to [11].

1.2.1 The language COLD

The backbone of SPRINT is the formal specification and design language COLD. COLD is an acronym for Common Object-oriented Language for Design and can be described as a state-based, model-oriented specification language which is based on the formal language COLD-K

[5, 6]. Actually, the language COLD as we describe it here is the user-oriented version COLD-1. However, to keep the overview simple, we will just write COLD when we refer to COLD-1. COLD supports several styles of describing a design such as algebraic specifications, axiomatic specifications, inductive definitions, specifications using pre- and postconditions and abstract algorithmic descriptions.

Because COLD was designed to be a general-purpose language, it has no predefined types, such as natural numbers or sets. Instead, it has *observers*, such as functions and predicates, and *transformers*, such as procedures. Both have as their domain *sorts*, which are the denotations of the dataypes. The transformers specify *transitions* between states, while each observer has a specific value that may change from state to state. The observers and transformers are grouped in *classes* which are the semantic objects of COLD descriptions. A class is denoted by means of a *specification component*. We will give a first practical introduction to COLD in section 1.3.

Modular design

An elaborate export and import mechanism allows for design in the large. Basically, a (specification) component can export certain observers and transformers, while a component can import exported observers and transformers from other components. In this way, a design can be divided in parts, and suitable parts can be candidates for reuse in future designs.

IGLOO library

The library IGLOO (Incremental Generic Library of Objects) contains a number of standard components, which are amongst others used to describe the basic data types. With the use of the import mechanism, any component can have these components at its disposal.

SPECICOLD

A subset of COLD, tailored to the use of pre- and postcondition techniques, is the language SPECICOLD [9]. It has been introduced to allow newcomers in the field of formal techniques to use the common concepts of pre- and postconditions without the burden of learning the other techniques present in COLD.

PROTOCOLD

Another subset of COLD, PROTOCOLD, supports executable specifications. These specifications are rule-based: they define state-transition rules. They can be executed because they assume that the imported components have some defined and executable implementation, for example written in C. It uses the information specified in the C header files to execute the specification. In this way, it allows easy and fast prototyping of a design.

1.2.2 The component concept

An important concept in SPRINT is the *component*. A component is an abstract entity which can have as attributes e.g. documentation, a formal specification and an implementation in a programming language. In this thesis we focus on the use of the specification part of a component, so we will use the word component whenever we mean the specification part of a component.

Generic components

As mentioned, a component is an abstract entity. So is the *generic component*, which is a component with parameters. In COLD it is allowed to write *parameterized* class descriptions. A generic component can be *instantiated* by providing actual parameters. Restrictions on these actual parameters can be expressed by means of a *parameter restriction* in COLD. The formal

basis for the instantiation of parameterized components is the $\lambda\pi$-calculus, developed by Feijs, which is described in [4].

Higher-order components

A *higher-order component* is a component that has a variable number of items in its signature. This variability is not directly expressible in COLD, so there has to exist a seperate component for each needed number of signature items.

Unfolding components

A component can be *unfolded* by including the definitions of the functionality of the imported components in the class-section of the unfolded component. The advantage of unfolding is that a number of checks on the imported components is not necessary anymore, so the analysis of the design with the tools can be faster and easier.

1.2.3 Tools

In the SPRINT-method, as in any industrial software engineering method, there is a need for tools. Among these are e.g. a syntax-checker and a type-checker, which are shortly described in [15]. In [14] a tool is described that allows the user to instantiate in an easy way frequently used components from the standard library IGLOO.

1.2.4 Summary

From the above we see that the SPRINT method indeed integrates specification with (SPECI)COLD, prototyping with PROTOCOLD and reusability with the IGLOO-library. As this is only a global introduction, we cannot discuss the details of this integration. However, the SPRINT method has been put to effective practice at Philips.

1.3 The specification and design language COLD

Introducing the language COLD with all its aspects and theory is easily a subject for a book, for example [5]. However, we hope to give a practical introduction to the basic concepts of COLD, relevant to this report, with the examination of the IGLOO library specification for TUP2, taken from [15]:

```
COMPONENT TUP2[Item1, Item2] SPECIFICATION

ABSTRACT
  ITEM1,
  ITEM2

EXPORT
  SORT Tup2
  FUNC tup : Item1 # Item2 -> Tup2
  FUNC proj1 : Tup2 -> Item1
  FUNC proj2 : Tup2 -> Item2

CLASS

  SORT Tup2 DEP Item1, Item2
```

```
  FUNC tup : Item1 # Item2 -> Tup2

  DECL t       : Tup2
     , i1, j1 : Item1
     , i2, j2 : Item2

  PRED is_gen : Tup2
  IND  is_gen (tup (i1, i2))

  AXIOM
  {TUP1} tup (i1, i2)! ;
  {TUP2} tup (i1, i2) = tup (j1, j2) => i1 = j1 AND i2 = j2 ;
  {TUP3} is_gen (t)

  FUNC proj1 : Tup2 -> Item1
  IND  proj1(tup(i1,i2)) = i1

  FUNC proj2 : Tup2 -> Item2
  IND  proj2(tup(i1, i2)) = i2

END
```

This specification describes a *sort* Tup2, which can be viewed as a data-type, and its associated functionality, in terms of the functions `proj1`, `proj2` and `tup`.

We can distinguish three main sections in each component: an `EXPORT` part, containing those sorts and operations that other components may import, an `IMPORT` part, which is missing here but which is used to indicate what sorts and operations may be used in this specification and in which components they are defined and a `CLASS` part, that is private to the specification and that may contain the definitions of the exported functionality.

1.3.1 Class

We will discuss the `CLASS` section first. The function `tup` in this case is a *constructor* function, which has the functionality to take two arguments from the sorts `Item1` and `Item2` and to map these to a corresponding tuple.

Note that by stating the functionality alone we have no idea how this map is defined. The *axioms* `TUP1` and `TUP2` are meant to specify this. For example, `TUP1` specifies that the function `tup` should be defined for all elements of `Item1` and `Item2`. Definedness is indicated by the `!` after `tup(i1,i2)`. That `TUP1` should hold for all elements is indicated by the declaration of `i1` and `i2` as arbitrary elements of sorts `Item1` and `Item2`. (`DECL i1 : Item1, i2 : Item2` abbreviates the use of `FORALL i1:Item1, i2:Item2` in each definition where they are referenced)

The second axiom, `TUP2`, specifies that each distinct pair `(i1, i2)` is mapped onto a distinct element of `Tup2` by means of the function `tup`.

Axioms

As the word suggests, axioms should hold in all states. The expression following the keyword `AXIOM` is called an *assertion*, built from propositional and first-order logic connectives, user-defined predicates and built-in predicates. The connective `;` is defined as a low-priority `AND`.

An example of a user-defined predicate is the predicate `is_gen`, which is defined on the domain `Tup2`. It is *inductively* defined on all `Tup2` which have been built with the constructor function `tup`.

However, just defining the predicate is not enough. An axiom TUP3 is added to state that is_gen should hold for all t that are elements of Tup2. In this way Tup2 contains precisely the wanted elements.

An example of a built-in predicate is the ! which, placed after a term, indicates definedness of the term.

Inductively defined functions

As opposed to the axiomatic specification of tup, we see that proj1 and proj2 are inductively defined by means of an assertion. They are defined to be the *least* function satisfying that assertion, provided it exists.

Dependent sorts

We see the keyword DEP after the sort name Tup2. This keyword indicates that Tup2 may only change if the sorts Item1 or Item2 change. By the way, the approach taken in COLD is to allow only *growing* sorts, so the verb "change" should be read as "grow".

States

This introduces us to the concept of *states*. In COLD, functions, predicates and sorts may be variable, which is indicated by using the VAR keyword as their definition. This means that the values of these functions and predicates may change from state to state. If the DEP keyword was missing in this specification, we would have an example of an *algebraic* specification. The difference with the so-called *state-based* specifications, is that all sorts, functions and predicates are fixed, i.e. they cannot be modified. Actually, the approach taken in COLD is to view algebraic specifications as a special case of state-based specifications: they have only one state.

The state is not explicitly present in the specification, it is implicitly present by the current values of the predicates, functions and sorts. A state-transition is described with *procedures*, which are not present in the above example. A procedure, indicated with the PROC keyword, describes exactly how the functions, predicates and sorts are modified.

1.3.2 Export

TUP2 exports most of its functionality, except the auxilary predicate is_gen. The reason for this is that is_gen is used only to specify the *minimality* of the sort in terms of the constructor function. In other words, is_gen holds for all t of Tup2 and would be equivalent to using TRUE. A component that imports TUP2 can reference any of the listed functionality. If no EXPORT section is given, then all the defined functionality is exported.

1.3.3 Abstract

We still do not know where and how Item1 and Item2 are defined. We do know however the *restrictions* that they must satisfy. These resctrictions are given in the components indicated by the ABSTRACT keyword. The restrictions in this case are simple, here follows the *abbreviated* component ITEM1.

```
LET ITEM1 :=

CLASS

  SORT Item1 VAR

END
```

A similar abbreviated component exists for ITEM2. Note that this component poses no restrictions at all on the sort Item1, due to the absence of axioms and the VAR keyword after the sort name, which indicates that the sort may grow. In other words, any sort satisfies this restriction.

1.3.4 Import

Although missing here, in an IMPORT section the components are listed from which this component can use the exported functionality. It follows the export section, so imported functionality can also be exported again by this component.

We give an indication of an IMPORT section by specifying another component that imports TUP2:

```
COMPONENT COORDINATE SPECIFICATION

EXPORT

  SORT Coordinate
  FUNC x_axis : Coordinate -> Nat
  FUNC y_axis : Coordinate -> Nat
  FUNC coordinate : Nat # Nat -> Coordinate
  FUNC origin : -> Coordinate
  % etc...

IMPORT

  NAT,
  TUP2'[Nat, Nat] RENAMING Tup2 TO Coordinate,
                           proj1 TO x_axis,
                           proj2 TO y_axis,
                           tup TO coordinate
                  END

CLASS

  FUNC origin : -> Coordinate

  AXIOM origin! ;
        origin = coordinate (0, 0)

  % etc...

END
```

This component specifies the sort Coordinate, which is a model for raster points in the upper-right quarter of the plane. We use the comment facility % to indicate that there may be more interesting functions to be specified.

Importing

Our attention is devoted to the IMPORT-section. We see that two components are imported: NAT and TUP2'[Nat, Nat]. NAT is a component specifying the sort Nat, which models the natural numbers.

Copying We see that a quote symbol is added immediately after TUP2. This quote indicates that not the component itself but rather a literal copy of it is to be imported. The reason why we need a copy lies in the fact that each function, predicate and sort must have a unique origin where it is defined. If we were to use Tup2 in another part of the design and we didn't make the copies, then the origins of the two tup functions would be the same which could introduce inconsistencies. So, to be on the safe side, we make a copy here.

Instantiation The square bracket part ([Nat, Nat]) indicates that the *actual* parameters for TUP2 are the sorts Nat and Nat. They are to replace the *formal* parameters Item1 and Item2. The parameter restrictions in TUP2 require the parameter Nat to be a sort that may be variable. This is indeed the case. Now, we can think of the imported copy of TUP2 as a TUP2 in which all occurences of Item1 and Item2 are replaced by Nat.

Renaming The import mechanism allows imported names to be renamed. These new names can then be used in the export and class sections of the component.

The resulting functionality In this way we have specified a suitable model for our coordinates without having to give the definitions of the functions x_axis etc. explicitly.

Concluding

We hope that this very rough introduction allows the reader to understand the terms and problems expressed in the following section.

1.4 The Assignment

We give here the description of the assignment

> "A library of standard components exists to be used in SPECICOLD designs. This library contains e.g. magnitudes (NAT, INT, RAT), enumerated datatypes (ENUM1, ENUM2, ···), structured datatypes (SEQ, BAG, SET, TUP1, ···) and many more. This library is called IGLOO, the Incremental Generic Library Of Objects. Although usage of this library is well known by SPRINT-developers, there are some disadvantages to this use of the library:
>
> - Some ranges of similar components cannot be taken as instantiations of one generic component (e.g. ENUM1 ··· ENUM49), because the number of items in their signature is variable.
> - The use of the copy-operator, renamings and parameter-instantiation complicate the development of analysis-tools. For example, a defining occurrence and an applied occurrence need not have the same name. By comparison : PROTOCOLD contains only import en export sections.
>
> An alternative approach in which the specification components are generated by a generator needs to be examined. Ultimately, all IGLOO-use could be replaced by the use of this generator."

Furthermore, a number of requirements were given:

Generation

Given a generic expression and arguments provided by the specifier the system instantiates the generic expression by mapping the arguments to the parameters of the generic expression.

Instantiation Semantics
The instantiation process should have a well-defined semantics.

Scope
The system should allow instantiation of at least the most important IGLOO library components.

Modification
When arguments were previously used to instantiate a given generic expression, the system will allow the specifier to modify these retrieved arguments, so the system can re-instantiate the component with the modified arguments.

Options
Among the arguments given by the specifier are options, which can be used to indicate choices in the way a generic expression is to be instantiated.

Generic Library
A number of generic expressions can be constructed and added to the system.

Private Library
The output of the system supports the construction of a private library of instantiated generic expressions.

SPRINT
The system is usable in the SPRINT software development environment.

Batchmode
The system supports a batchmode-like operation that allows the automatic instantiation of a given generic expression with a given set of arguments.

GUI
The system provides the user with a consistent GUI which contains sufficient elements to let the specifier indicate his arguments to the system.

Ease-Of-Use
It should be relativily easy to use the system, as compared with the current way of instantiation.

1.5 Description of the solution

From the problem statement it is clear that a generator tool should be developed. This generator is presented to the users, who can call this generator with certain arguments. The generator then processes these arguments and creates the result, which should be a component, at least for now the specification part of that component.

1.5.1 A language

Because of the large number of suitable generic components and the different ways in which they can be instantiated, we see the need to develop a language which can be used to describe the aspects of the instantiation. All generic components can be rewritten in this language. The arguments that the user provides for the generator are somehow applied to these rewritten components by the generator with as a result the instantiated component. These arguments can be provided on the command-line or via a graphical user interface.

1.5.2 Structure of the language

In [13] a language for defining virtual libraries of infinite data types for LOTOS, a protocol specification language, is discussed. We have examined whether this language could be tailored to this specific problem. In chapter 3 we will present our version of this language and call it *TEMPL*, which stands for TEMPlate Language or, more general, The Easy Macro-Processing Language. In TEMPL generic components can be written.

1.5.3 Another language

For the instantiation of a TEMPL component there is the need for some kind of access language, by which the user can indicate arguments specific for the instantiation. We will call this language the *instantiation language* or the instantiation arguments.

1.5.4 The system

The generator tool itself can be viewed as a system, consisting of a *parser*, for parsing a TEMPL component, an instantiator, for the actual creation of the instantiated component and a *mapper*, which applies the instantiation arguments to the TEMPL. Furthermore, there are the TEMPL descriptions of the components and an user interface. We will call this system *TEMPLES*, which stands for TEMPL Expression System.

In chapter 2 we will describe how the system developed from the requirements and the specific details of our development method.

Chapter 2

The Design of TEMPLES

In our design of the system, we have strongly focused on the analysis and refinement of the requirements. We have chosen to combine the phases of requirements analysis, design and specification of the traditional waterfall model for software engineering together in our approach by describing the *transformation* of the requirements as guidelines for the design. This process ultimately and naturally leads to a formal specification. The reason why we are able to design in this manner lies in the availability of the customer, our use of standard software technology for the implementation of the system and the fact that there are no extraordinary requirements. The testing of the system is described to some extent in appendix F. The unit integration phase of the waterfall model is not described in this document as it is in this case a straightforward reversal of the top-down design components.

After an introduction to the method, we give in this chapter an example of the design of some parts of the system using this method in section 2.2 and we conclude with a comparison of our method with another design method found in the literature, essential systems analysis in section 2.3.

A final system description can be found in appendix C as a users manual.

2.1 Activities before the design

First, we have thoroughly examined the documents related to the problem. This allowed us to gain insight in the problem domain. We were then able to state the requirements more precisely, as we have seen in section 1.4.

Furthermore, we investigated the tools available for the implementation of the system. With this information we were able to sketch the first solution of the problem described in section 1.5.

2.1.1 Using requirements in the design

We have seen that our problem statement given in section 1.4 is not very explicit. Instead, a number of *requirements* have been given which the system should satisfy. These requirements fall in the category of *functionality* requirements. In our design process, we will divide and refine these requirements in ever more detail to aid in the definition of the systems implementation.

Furthermore, current technology and tools or in general the *resources* at our disposal restrict the solution to a *feasible* one. These requirements can be seen as feasibility requirements, or *non-functional requirements.*

We used both kinds of requirements in the actual design process. If at some point the functionality requirements clashed with the non-functional requirements then we reformulated the functionality requirements with the aid of the customer at a higher level in the design process.

2.1.2 A data dictionary

Each system and each designer introduces its and his own concepts and terms to the unwary reader. The context in which the terms introduced by the designer are used, is determined by the system. The explicit mentioning of a kind of *dictionary* reminds the reader of a new context for terms he is familiar with and on the other hand, is a promise of the writer that he shall use the terms according to their meaning in the dictionary. That is way we will give an explicit dictionary before we start designing the system. The data dictionary is given in appendix A.

2.1.3 Top-down design

For the system, we have chosen a top-down design method. This choice was motivated by the fact that the system was new, so it wasn't possible to use previously defined components, and the clear input-output behaviour of the system, which allows easy divisions at each stage of design.

So, we started off with the requirements and a solution in general terms and tried to refine and split the requirements as we divided the solution in partial subsolutions.

At each of these stages, we examined the components by

- Stating the requirements applicable to the component in the formality needed for the current stage of design.
- Verifying that these requirements are indeed derivable from other requirements and that they do not conflict with other requirements.
- Describing the component that satisfies these requirements.
- Verifying that the component satisfies these requirements.
- Checking the feasibility of the component.
- Dividing the component into suitable subcomponents to be defined at the next stage.

The design of an arbitrary component need not necessarily go through all these steps, as some of them may be trivial.

Finally, we arrived at the stage where all components are described in such detail that they can be implemented in a language for the machine. Then we reintegrated all components to one system. This reintegration allowed us to validate the more formal stages of the design from the bottom up against the informal ones from the top down.

In our view, designing the system in this way allows the reader to follow the design process closely and paves the way for the creation of technical documentation tightly bound to the design process, which in its turn allows for future modification of the system in a relative easy way.

2.2 Design example

We choose not to give the design of the entire system. Instead, we choose the design of some interesting components that will clearly demonstrate our design process.

We use a "breadth-first" approach to describing our example components. To keep an overview of the locations of the components we have included some figures. In these figures, the components are indicated by a rounded box, and the components further divided in this section are indicated by a fat box.

We divided the initial solution described in subsection 1.5 as follows:

- Information components
  These components define the external information that is input to or output of the system.

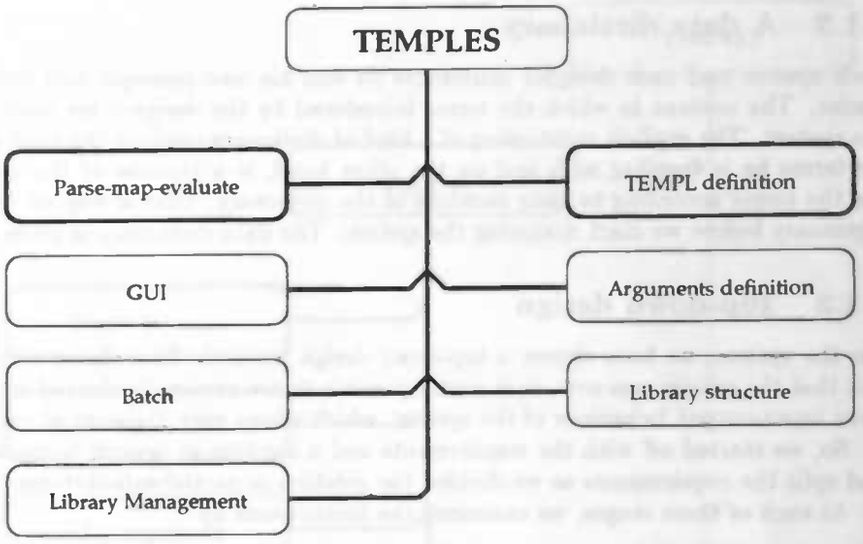


Figure 2.1: TEMPLES design: Finding the main components

- ⋆ A *TEMPL definition*, which should define syntax and semantics of the TEMPL language. This component is described to some detail in subsection 2.2.2.
- ⋆ An *arguments definition*, which should define syntax and semantics of the instantiation language. Some requirements are given in subsection 3.1 in chapter 3.
- ⋆ A *library structure*, containing inputs and outputs of the system. We will not describe this component any further.

- Algorithmic components
  These components describe what should be done with the above information and how this should be done. We have included the design of the first (parse-map-evaluate) component in this section but excluded the other algorithmic components.
  - ⋆ A *parse-map-evaluate* component, which maps arguments to a TEMPL expression and creates the instantiated component. This component is described in detail in subsection 2.2.1.
  - ⋆ A *GUI* component, which allows for interactive use of the system.
  - ⋆ A *batch* component, which allows for batchmode interaction with the system.
  - ⋆ A *library management* component, that stores and retrieves information in and from the libraries.

Note that we introduce components here that are to some extent independent of each other. They are not fully independent, which implies that we should be careful with the definitions of interfaces between the components. A picture representing this division can be found in figure 2.1.

For this example we will now focus on the parse-map-evaluate component and the TEMPL definition component. This choice is motivated by the fact that the parse-map-evaluate component is more or less the core of the system and the fact that the TEMPL definition component is described fully in chapter 3.

We abbreviate the requirements as follows : **component-name.requirement-id**, in which **component-name** is the name of the component on which the requirement is directly applicable.

The **requirement-id** is a shorthand for the contents of the requirement. The top-level requirements given in section 1.4 have no associated component name, they are directly referenced by their requirement-id. This notation allows us to refer easily to the requirements while still giving some insight in the purpose of the requirement.

2.2.1 The parse-map-evaluate component

As a part of the top-level design, this component is the critical part of the system and should be designed with care. The most important requirement relating to this component is *Generation*, defined in section 1.4.

Requirements for the parse-map-evaluator In the requirements, we will use the abbreviation PME for parse-map-evaluator.

PME.Syntax
The PME should report clearly any occuring syntax errors in the TEMPL expression.
Derivation
From requirement *Ease-of-Use*. It is included in order to allow the TEMPL developer to see clearly what kind of errors made an instantiation impossible.

PME.Arguments
The PME should report errors and/or warnings if the mapping from arguments to parameters went wrong.
Derivation
See *PME.Syntax* for its derivation.

PME.Instantiation
The instantiation process should have a clear and unambigous formal semantics.
Derivation
This is a strong refinement of requirement *Generation* and follows from *TEMPL.Semantics*, given in subsection 2.2.2.

PME.Imports
The PME should process the import list of the TEMPL expression by reinstantiating these imported components
Derivation
The independence of TEMPLES of the IGLOO library makes it necessary for the system to reconstruct all needed library components that are imported into the TEMPL expression at hand. This is also required by *Private Library*.

Conflicting requirements We can see that *PME.Imports* and *TEMPL.Generality* (see subsection 2.2.2 conflict, in the sense that the PME should be able to extract from the TEMPL expression the import section. However, as TEMPL is supposed to be a general language, this can not be done. Furthermore, reinstantiating these imported components would require that some provision must be made in the language to allow for a recursive call to the PME with arguments. Because the parameters used in the TEMPL expression can be chosen freely (there are no built-in options) this cannot be done in the general case. We reformulate *PME.Imports* as follows:

PME.Imports
The PME should attempt to extract the import list of the instantiated TEMPL expression and report any found imported components.
Explanation
The task of instantiation of these imported components is thus left to the specifier.

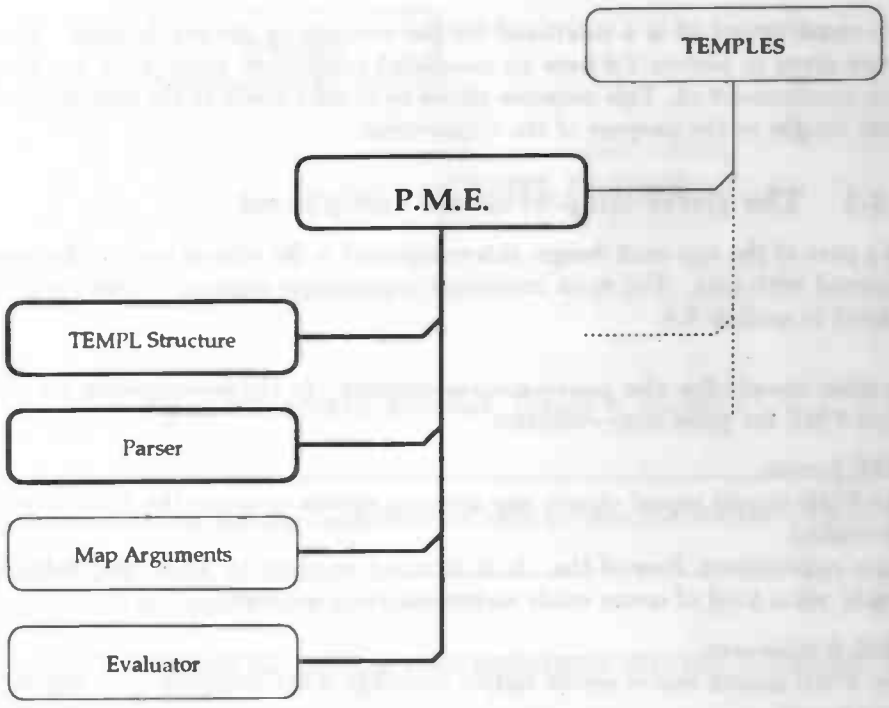


Figure 2.2: Parse-Map-Evaluate: Easily found components

Derivation
See discussion above.

Design of the parse-map-evaluate component This component is directly related to the TEMPL definition component. Its purpose is to translate a given TEMPL expression name with arguments to an instantiation, according to the semantics of the TEMPL language.

Dividing the Parse-Map-Evaluate component There exists a natural division for this component, namely

- A *Parser* component, described in subsection 2.2.4, which parses a TEMPL expression to a yet undefined TEMPL structure.
- A *Argument Map* component, described in subsection 2.2.5, which substitutes the actual (argument) parameters for the formal parameters of the TEMPL structure.
- An *Evaluate* component, described in subsection 2.2.6, which evaluates the TEMPL structure to its instantiation.

The intermediate *TEMPL structure* mentioned will be defined in subsection 2.2.3. A pictorial representation of the division of the PME component can be found in figure 2.2.

Feasibility Feasibility of this component is of course very important. However, if the language does not introduce too strange, new constructs, feasibility (*Time*) depends on the availability of a powerful parser generator tool. Fortunately, this tool is present and is called Elegant. A short description of this tool is given in appendix B, for a more thorough description we refer to [1] and [8].

2.2.2 The TEMPL definition component

This component should specify the language constructs present in TEMPL, the exact syntax of TEMPL and its semantics. Furthermore, a manual for the TEMPL developer should also be created as part of this component. See figure 2.3 for the conceptual location of this component.

The previous requirements to be satisfied by this component are *Generation*, *Options* and *TEMPL Library*. The following requirements are found as a result of refining these requirements:

TEMPL.Generality

The TEMPL language should be usable for COLD and other formal languages.

Derivation

This requirement is actually a consequence of our initial investigations in finding suitable language constructs for genericity. The idea is, that the language can be viewed as a *macro preprocessing language*. Note that this requirement does not conflict with *SPRINT*, because the system could do some postprocessing on the instantiation for this specific purpose.

TEMPL.Parameters

The parameters used in TEMPL expressions should support an option mechanism, a simple parameter mechanism for simple parameters and an extended parameter mechanism for many valued parameters.

Derivation

The option mechanism is required by *Options*, the second mechanism follows from the fact that the language should provide at least constructs that are available now and the third mechanism is a consequence of the new facilities needed for the extension of the genericity principle.

TEMPL.Correctness

The language should provide a means to ensure that the instantiation is meaningful.

Explanation

The flexibility of the option mechanism allows meaningless instantiations, unless the arbitrary selection of options is somehow prohibited.

Derivation

This requirement can be derived from *Generation* and *Private Library*. If the specifier is to be able to build a private library then this library should at least be as meaningful as the current IGLOO library.

TEMPL.Semantics

In order to write meaningful TEMPL expressions, the language should have a clear and well-defined semantics.

Explanation

In relation with *TEMPL.Correctness* it is necessary to define the meaning of a TEMPL expression precisely or to indicate why this meaning cannot be given.

Derivation

It is of course common sense to define the semantics of any language. In this case it is of specific importance, because of the potential use of TEMPL expressions in the formal method of SPRINT.

TEMPL.Syntax

The syntax used for TEMPL should allow format preserving instantiation and maintain a clear overview of the underlying source language.

Explanation

Because the results of the instantiation must be readable in a normal way by the specifier, the format of the instantiated expression should already have been indicated in the TEMPL expression somehow. The TEMPL developer on the other hand must be able to see the structure of the underlying source expression in a relative easy way.

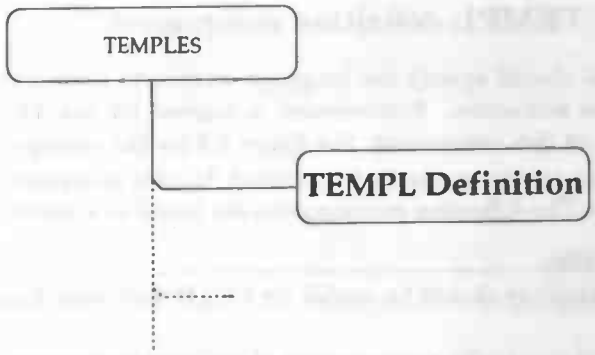


Figure 2.3: TEMPL definition: Details in chapter 3

Derivation
The added requirement here is really readability for both the specifier and the TEMPL developer.

TEMPL.Ease-of-Use
The language should be easy to learn and easy to use.
Derivation
This requirement follows from the common sense requirement that any tool should lighten the burden of specification.

We will not describe the design of the language in detail here, and refer to chapter 3 for a more thorough treatment of TEMPL.

2.2.3 The TEMPL structure component

This component should hold information on the abstract syntax of the TEMPL expression, its declared options and parameters and its constraints. The abstract syntax of TEMPL is described in chapter 3.

The following requirements can be given for the TEMPL structure component:

TEMPL Structure.Contents
The contents of the TEMPL structure should reflect the information content of the corresponding TEMPL expression by holding information on the abstract syntax of the TEMPL expression, its declared options and parameters and its constraints.
Derivation
Because the TEMPL structure is to be used as an intermediate replacement of the TEMPL expression, and is subsequently used in the Map Argument and Evaluate component, the essential information of the TEMPL expression must be retained.

TEMPL Structure.Well-formed
The TEMPL Structure corresponding to a well-formed TEMPL expression should have the following form:

- **Options**
  For each option a structure containing the identifier of the option, the help tekst belonging to this option and an indication that this option is not (yet) selected.

- **Parameters**
  For each parameter a structure containing the identifier of the parameter, the declaring option of that parameter and the help tekst belonging to this parameter.
  Furthermore, we state

  - ⋆ **Singular**
    For each singular parameter the argument value should be present, being initially the identifier of the parameter.

  - ⋆ **Sequence**
    For each sequence parameter the lowerbound of the parameter, the identifier for the upperbound of the parameter, an indication whether this parameter determines its upperbound and an initially empty list of argument values for this parameter should be given.

- **Dimensions**
  For each dimension a structure containing the identifier and the value, initially zero, of the dimension.

- **Constraints**
  An abstract version of the set of constraints, by which we mean that the semantic meaning of a syntactical constraint is captured in an abstract structure.

- **Expression**
  An abstract version of the actual expression, by which we mean that the semantic meaning of a syntactical expression is captured in an abstract structure.

Derivation
The well-formedness of this intermediate structure is necessary to allow the formal definition of the other PME components that use this structure. The implicit subdivision into components of the TEMPL structure follows from the language elements present in the TEMPL definition.

Design of the TEMPL structure From the requirements we see that the structure is easily divided into a number of substructures, as shown in figure 2.4. We propose to define these substructures in separate components. Note that we now have enough precise requirements for these components that we can attempt to define them formally. In this example, we will focus on a derivation of a formal specification of the parameters component, which is described in subsection 2.2.7.

Feasibility Because the Parser will be implemented in Elegant, feasibility requires us to implement the above TEMPL structure also in Elegant for the following reasons:

- The interface between Elegant and other languages (C) is for stand-alone functions easy, but for the more complex data structures here rather difficult.
- Maintainability and a high level of abstraction are guaranteed.

2.2.4 The Parser component

See figure 2.2 for the conceptual location of this component.

Requirements for the Parser These requirements are based on the TEMPL language elements and the TEMPL structure, the intermediate result in the PME component.

Parser.Correct Syntax
If the TEMPL expression is syntactically correct, then the corresponding TEMPL structure should be created according to *TEMPL Structure.Contents*.

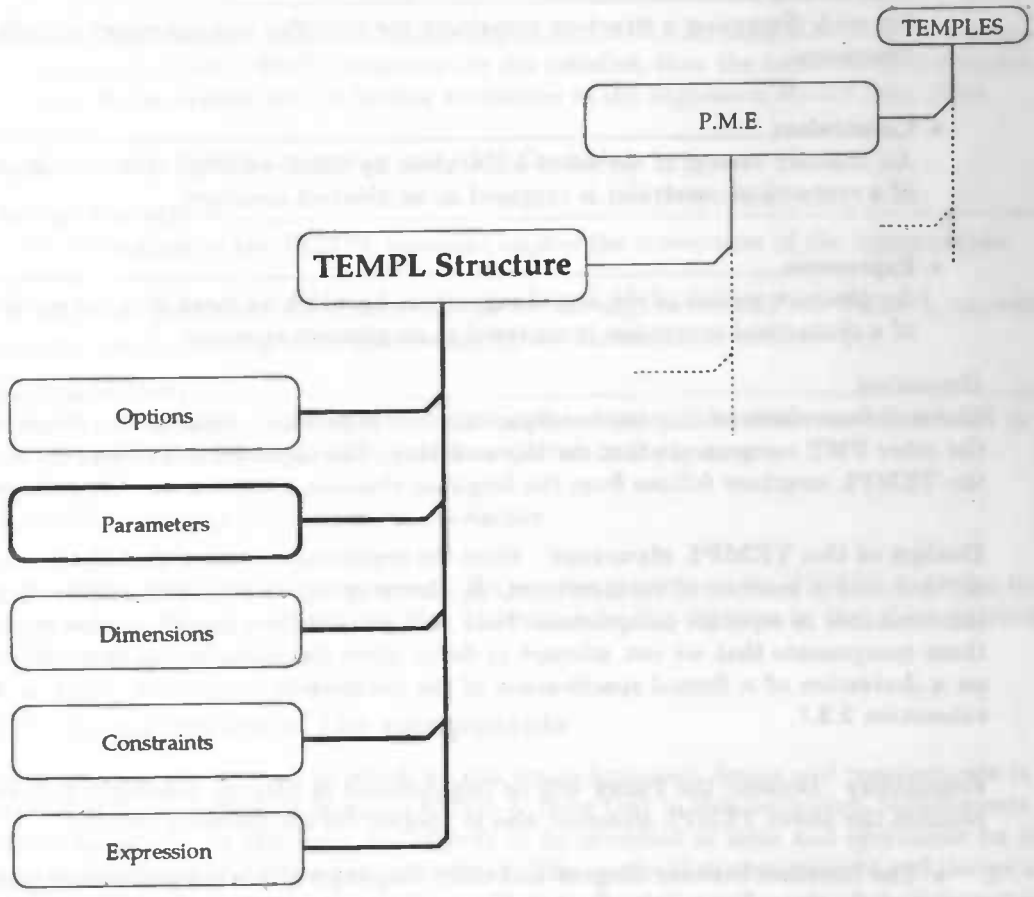


Figure 2.4: TEMPL structure: Suitable language abstraction

Derivation
The Parser must make sure that the information of a correct TEMPL expression is stored correctly in the TEMPL structure. This requirement is part of *Generation*, which we can split up now because of our division of the PME.

Parser.Incorrect Syntax
If the TEMPL expression is syntactically incorrect, then the syntax violations should be reported to the system.
Derivation
This requirement is a direct refinement of *PME.Syntax*.

Parser.Errors
If identifiers are redeclared, or if undeclared identifiers are used, then these errors should be reported to the system.
Derivation
These errors are not really syntax errors, because syntax is usually context free, and these errors are derived from context. However, the Parser should report them, in a separate category, as a result of the instantiation process (See e.g. *GUI.Instantiation Process* and *Generation*).

Parser.Resources
The Parser should be able to build a resources structure, which can be used to build the TEMPL specific GUI.
Derivation
This is a consequence of the decision to let the GUI interaction be a special case of the batchmode interaction. To allow for GUI interaction, the system needs to now what information to display.

Design of the Parser component In the design of a Parser, we usually write the grammar of the language in BNF-form, which is then used as a guideline for actions to be taken on recognition of one of the terminal symbols. The Elegant tool provides a systematic way of constructing a Parser. Part of this process is described in section 3.5.3.

2.2.5 The MapArgument component

See figure 2.2 for the conceptual location of this component. After the internal arguments have been set by the batch component, these arguments can be mapped onto parameters and elements in the TEMPL structure.

The following requirements can be given for the MapArgument component:

Map.Correctness
A well-formed argument specification mapped onto a well-formed TEMPL structure should result in a well-defined TEMPL structure.
Derivation
This requirement is needed to allow for a formal definition of the semantics of the instantiation process, as such it can be derived from *TEMPL.Semantics*, *Instantiation Semantics* and *Arguments.Parameters*.

Map.Incorrectness
If the argument specification is not well-formed or if the mapping from a well-formed argument specification onto a well-formed TEMPL structure could not be performed successfully, then these problems should be reported to the system
Derivation
This is a refinement of *PME.Arguments* and is needed for *GUI.Instantiation process*.

Design of the MapArgument component Because the arguments provided by the batch component must somehow be manipulated, we propose to store the arguments into an intermediate arguments structure, which allows a better and simpler way of describing the mapping process.

2.2.6 The Evaluation component

The evaluation component is derived from the PME component, as shown in figure 2.2. After the mapping of the arguments, we can assume that the arguments are well-defined and that we can apply the expression to these arguments, if of course the constraints are satisfied.

The following requirements can be given for the Evaluation component:

Evaluation.Constraints

If the constraints in the TEMPL structure are not satisfied, then the unsatisfied constraints will be reported to the system and no further evaluation of the expression should take place.
Derivation
This follows from *TEMPL.Correctness* and *PME.Constraints*.

Evaluation.Correctness

The well-definedness of the TEMPL structure implies the correctness of the instantiation.
Derivation
The system has performed all possible checks on the arguments and the TEMPL expression. The system can do nothing more than assuming the instantiation correct.

Evaluation.Imports

As a result of the instantiation process, the imported components that are mentioned in the import section of the instantiation should be reported to the system.
Derivation
From *PME.Imports* and *GUI.Instantiation semantics*.

Design of the Evaluation component We propose to evaluate the TEMPL structure to one string, which can then be written to a file. The reason for this decision is, that the constructs in TEMPL result in strings.

2.2.7 Specification of the components

We now come to the moment in which we can try to formulate design and requirements in the formalism of the specification language COLD-1. Note that in this formalism, requirements can be stated as axioms, or theorems, the objects to be modelled as sorts and operations on these objects as functions, predicates and procedures. This means that requirements and design start to mingle at this stage. In this example we will give a formal specification of the parameters of the TEMPL structure (see figure 2.4) and the parser component (see figure 2.2).

The string parameters of the TEMPL structure

We need some basic components in order to describe the design of the TEMPL structure. We take the IGLOO library components `NAT` and `STRING` as defined.

Furthermore, we create a new basic component, `IDENT`, which describes the sort `Ident`, as a model for the identifiers in the TEMPL structure. The reason for this basic component, is that the identifiers for the different language elements may overlap. By defining the properties of `Ident` in a separate component we can use `Ident` as a basis for our identifiers by importing `IDENT` in all our identifier components.

In this example we will focus on the specification of the parameters. We state the following requirements for our parameters:

Parameter.Unbounded
There is an unbounded number of parameters at our disposal.

Parameter.Distinct
Distinct parameter denotations from the TEMPL expression must be mapped onto distinct parameters in the TEMPL structure.

Parameter.Declare
Parameters may only once be declared.

Parameter.Init
Initially, there are no declared parameters.

Parameter.Kinds
Two kinds of parameters exist: singular and sequence parameters. Each declared parameter is one of these two kinds.

Parameter.Singular
Each singular parameter can be given at most one value.

Parameter.Sequence
Each sequence parameter can be given a range of values. This range depends on the lowerbound and the value of the upperbound of the parameter.

Parameter.Option
An option declares at most one parameter.

Parameter.Dimension
Each dimension is determined by precisely one sequence parameter.

Specification of the parameter component We will first describe the exported functionality:

```
COMPONENT PARAMETER SPECIFICATION

EXPORT

  SORT Parameter

  FUNC prefix : Parameter -> String
  FUNC parameter : String -> Parameter
  FUNC option : Parameter -> Option
  PRED is_singular : Parameter
  PRED is_sequence : Parameter
  PRED is_declared : Parameter

  FUNC value : Parameter -> String
  PROC set_value : Parameter # String ->
  PROC declare : String # Option ->

  FUNC lowerbound : Parameter -> Nat
  FUNC upperbound : Parameter -> Dimension
  PRED determines : Parameter

  FUNC value : Parameter # Nat -> String
```

```
  PROC set_value : Parameter # Nat # String ->
  PROC declare : String # Option # Nat # Dimension # Bool ->

  PRED wf
```

We import the following components:

```
IMPORT
  DIMENSION,
  OPTION,
  NAT,
  BOOL,
  STRING,
  IDENT' RENAMING Ident TO Parameter END
```

The component IDENT defines the procedure declare, which we can now use for our parameters and the predicate is_declared.

Now we can try to translate the requirements into an invariant. By our renaming of Ident to Parameter we have achieved already requirements *Parameter.Unbounded*, *Parameter.Distinct*, *Parameter.Init* and *Parameter.Declare*.

```
CLASS

  DECL pmr : Parameter
     ; s,t : String

  FUNC parameter : String -> Parameter

  AXIOM parameter (s)!
      ; s /= t => parameter (s) /= parameter (t)
```

For each string there should exist a parameter. The second part of the axiom indicates that this parameter is unique.

```
  FUNC prefix : Parameter -> String VAR
  FUNC option : Parameter -> Option VAR

  PRED is_singular : Parameter VAR
  PRED is_sequence : Parameter VAR
```

The above functionality is present for all parameters. As we will later see, they are only defined on declared parameters.

The following functionality is present for singular parameters.

```
  FUNC value : Parameter -> String VAR

  PROC set_value : Parameter # String ->
  IN   p,s
  PRE  is_declared (p)
     ; is_singular (p)
  SAT  MOD value (p)
  POST value (p) = s

  PROC declare : String # Option ->
  IN   s,o
```

```
PRE  NOT (is_declared (parameter (s)))
   ; is_declared (pmr) => option (pmr) /= o
SAT  MOD is_declared (parameter (s))
       , prefix (parameter (s))
       , option (parameter (s))
       , is_singular (parameter (s))
   ; declare (parameter (s))
POST is_declared (parameter (s))
   ; is_singular (parameter (s))
   ; prefix (parameter (s)) = s
   ; option (parameter (s)) = o
```

The following functionality is present for sequence parameters:

```
FUNC lowerbound : Parameter -> Nat VAR
FUNC upperbound : Parameter -> Dimension VAR
PRED determines : Parameter VAR
FUNC value : Parameter # Nat -> String VAR

PROC set_value : Parameter # Nat # String ->
IN   p, n, s
PRE  is_declared (p)
   ; is_sequence (p)
SAT  MOD value (p, n)
POST value (p, n) = s

PROC declare : String # Option # Nat # Dimension # Bool ->
IN   s, o, n, d, b
PRE  NOT (is_declared (parameter (s)))
   ; b => (is_declared (pmr) AND determines (pmr) =>
                                    upperbound (pmr) /= d)
   ; is_declared (pmr) => option (pmr) /= o
SAT  MOD is_declared (parameter (s))
       , prefix (parameter (s))
       , option (parameter (s))
       , lowerbound (parameter (s))
       , upperbound (parameter (s))
       , determines (parameter (s))
   ; declare (parameter (s))
POST prefix (parameter (s)) = s
   ; option (parameter (s)) = o
   ; lowerbound (parameter (s)) = n
   ; upperbound (parameter (s)) = d
   ; b => determines (parameter (s))
```

The following predicate, `wf` is used to indicate that the declared parameters are indeed defined for all the needed values. This predicate could be used as a precondition for an evaluate procedure.

```
PRED wf
DEF  is_declared (pmr) => is_defined (pmr)

PRED is_defined : Parameter
DEF  is_sequence (pmr) =>
       (FORALL i : Nat (( i >= lowerbound (pmr) AND
```

```
            i <= value (upperbound (pmr)) ) => value (pmr, i)!)
    ; is_singular (pmr) => value (pmr)!
```

We now come to stating the component invariant. We choose not to do so explicitly and define the invariant as a *theorem*, which is an assertion that should follow from the possible state transitions and the initial state. To this end, we first give an axiom that indicates the assertion that should hold in the initial state:

```
AXIOM INIT => (
        prefix (pmr)^
      ; option (pmr)^
      ; NOT (is_singular (pmr))
      ; NOT (is_sequence (pmr))
      ; NOT (is_declared (pmr))
      ; lowerbound (pmr)^
      ; upperbound (pmr)^
      ; NOT (determines (pmr))

THEOREM
  is_declared (pmr) => ( prefix (pmr)!
                       ; option (pmr)!
                       ; is_singular (pmr) XOR is_sequence (pmr) )
; is_sequence (pmr) => ( lowerbound (pmr)!
                       ; upperbound (pmr)! )
; is_declared (p) AND is_declared (q) =>
    ( determines (p) AND determines (q) AND p /= q =>
                       ( upperbound (p) /= upperbound (q) )
    ; p /= q => option (p) /= option (q) ) )

END
```

It is rather trivial to see that the procedures do indeed maintain the invariant.

Component PARSER

In the component PARSER the functionality of the parser is described.

```
COMPONENT PARSER SPECIFICATION

EXPORT

  PROC parse : ->

IMPORT
    TEMPL
  , EXPRESSION
  , PARAMETER
  , DIMENSION
  , CONSTRAINTS
  % etc...

CLASS

  PRED wf : Templ
```

```
PROC parse : ->
SAT  MOD the_expression
   ; declare ($)
  % etc...
POST wf (the_templ) => ( the_expression = expression (the_templ)
                       ; the_constraints = constraints (the_templ) )
   ; NOT (wf (the_templ)) => ( the_errors = errors (the_templ) )

END
```

The procedure `parse` is used to modify `the_expression` and `the_constraints` if `the_templ` is well-formed, otherwise it modifies `the_errors` to contain the errors of `the_templ`.

The predicate `wf` is not specified any further, it holds for all semantic and syntactic correct TEMPL expressions.

2.2.8 Implementation

Now we have described the components in such detail, we can try to implement them in a programming language. It is important to make sure that the implementations satisfy their specifications. When we combine implementations together, we should make sure that they satisfy the requirements at the higher level. This can be achieved by testing, of which we give examples in appendix F or by (in)formal reasoning.

We will not discuss the implementation effort in detail here and end our example of the design.

2.3 Comparing our design process

We will compare our design process with the design method of *essential systems analysis*.

Essential systems analysis was developed by McMenamin and Palmer [12], as an improvement to Structured Analysis. The method emphasises on the definition of the *essence* of the system. Two main parts of a system are distinguished: a *processor*, which performs actions on the system data stored in a *container*. The *essential* activities of a system are found by examining what the system should do if it were implemented with perfect technology. So, they can be defined as the activities of a perfect processor, which never makes mistakes, on a perfect container, which contains any amount of data that can be accessed instantly.

Essence We come to the following definition of essential activities:

> *Essential activities* are all the tasks that the system would have to perform even if it were implemented using perfect technology.

The essential memory is defined as follows:

> *Essential memory* consists of the data that the system would have to remember even if it were implemented using perfect technology

The essence of the system can be defined as follows:

> The *essence* of the system consists of the essential activities and the essential memory.

Incarnation The implementation of the system in less than perfect technology is called its *incarnation*. This incarnation includes the implementation of the essential activities as well as the additional data and operations needed to compensate for imperfect technology.

How to find essential activities Essential activities are triggered by an event and consists of a set of planned responses to that event, by which essential memory may be accessed. Two kinds of essential activities can be distinguished:

- *Fundamental* activities justify the existence of the system, such as producing new information from the system's essential memory.
- *Custodial* activities acquire, store and update the system's essential memory. They are essential, because they keep the system's memory consistent with the environment.

Modelling the system The essence of the system should be partitioned so it can be understood better. Both fundamental activities and essential memory are partitioned. The custodial activities are added if a particular incarnation is chosen. There are two ways of partitioning the model:

- Event-partitioning uses diagrams to display the system's entire respons to a single event, after which the system becomes idle.
- Object-partitioning groups the system's data elements to attributes of objects which are outside the system.

Designing the incarnation With the design of the incarnation, we can address issues such as cost, capacity, capability and fallibility of the system. As mentioned, in this phase the custodial activities and other memory routines, such as sorting or storing data are added.

The method One approach to design a system using this method is to start with an essential system and incarnation in general terms and to develop both models in parallel by adding details in a top-down way. The advantage of this approach is that the customer can validate the implementation as a prototype earlier in the design process.

A comparison Now we can compare our design process to the essential systems analysis method. The essential model is related to the requirements analysis that we have carried out. However, we have explicitly included feasibility checks in our process to prevent unachievable implementations. We also didn't make the essential activities explicit, although they are described by the requirements.

Our method introduced implementation dependent data as we went from the top down, as for example the resources structure is created as part of our model, but which is not essential data.

We did not build the essential and incarnation model separately, but achieved the implementation by carrying out the requirements analysis in detail.

Concluding, we say that this design method may have been interesting for the top of our system, especially by using the diagrams to give a clear overview of the activities of the system. For the remaining design phases, this method is not very useful for this system, as the limited set of activities and events are easy to capture in a few lines or pictures.

Chapter 3

The Language TEMPL

After our first investigations in the problem domain, we came to the conclusion that a language should be developed to describe the generic aspects of components. In this chapter we will first motivate our specific approach in which the language TEMPL is combined with COLD. Then we will describe three important aspects of this language. These aspects have been given by Schmidt in [16]:

- *Syntax* : The appearance and structure of its sentences.
- *Semantics* : The assignment of meanings to the sentences.
- *Pragmatics* : The usability of the language. This includes the possible areas of application of the language, its ease of implementation and use, and the language's success in fulfilling its stated goals.

Furthermore, we will give a well-documented example of a TEMPL expression for $\texttt{TUP}_n$, and conclude the chapter with some possibilities for the extension of the language.

3.1 Motivation of TEMPL

In this section we will describe how the language TEMPL evolved from the requirements. To this end we will state the requirements for the language as they evolved from the requirements analysis described in chapter 2.

TEMPL.Generality
The TEMPL language should be usable for COLD and also for other formal languages.

TEMPL.Parameters
The parameters used in the TEMPL expression should support an option mechanism, a simple parameter mechanism for simple parameters and an extended parameter mechanism for many-valued parameters.

TEMPL.Correctness
The language should provide a means to ensure that the instantiation is meaningful.

TEMPL.Semantics
In order to write meaningful TEMPL expressions, the language should have a clear and well-defined semantics.

TEMPL.Syntax
The syntax used for TEMPL should allow format preserving instantiation and maintain a clear overview of the underlying source language.

TEMPL.Ease-of-Use
The language should be easy to learn and easy to use.

Arguments.Parameters
Arguments should be unambiguously mappable to the corresponding parameters.

Arguments.Syntax
The syntax for providing arguments is simple.

As TEMPL was intended to be used as an extension to COLD, we first investigated in which way TEMPL could interact with COLD.

Two main approaches exist for extending COLD with macros:

- An extensional approach, in which COLD is extended with elements which allow especially the new constructs.
- A lexical approach, in which new macro-constructs are added which operate on strings built from the COLD syntax.

In the following subsections we discuss the implications of these approaches.

3.1.1 The extensional approach

This approach has as an advantage that readability of the TEMPL expressions is in principle assured. This is achieved by using terminology from the COLD world in the language or, conversely, by letting the new constructs be an extension to the COLD language itself. We would have to develop the language COLD-T, say, and define its meaning and well-formedness in relation to the kernel language COLD-K.

Unfortunately, certain constructs that we need, such as the optional inclusion of objects in the export signature, are not directly mappable to COLD-K constructs, let alone that they can be given a meaning in the COLD model. This would lead to a more constrained form of use of the language, conflicting with *TEMPL.Parameters* and *TEMPL.Generality*. Using an extended COLD syntax can be confusing to the specifier, because it brings again variety to the COLD language family. Furthermore, the need to parse an extended syntax of COLD could conflict with *Time*, because expressions of the general purpose language COLD are built from a large grammar.

3.1.2 A lexical approach

With this approach, new choices are possible regarding the way in which the constructs are used:

- Certain COLD language keywords could be used to aid in parsing the TEMPL expression.
- Macro-constructs could be used without any knowledge of the underlying text.

Using COLD-keywords in the text as guidelines for changing the way of instantiating a TEMPL expression is of course very useful. For example, if the COLD-keyword `IMPORT` is encountered, we can assume that each imported component in this section can be generated in uppercase[1]. However, when one adopts such a strategy so early in the instantiation of a TEMPL expression, then the language is tightly bound to COLD, which conflicts with *TEMPL.Generality*.

[1] This is not required by COLD, but a common project convention

In order to use the language as a macro language for many more formal languages, the evaluation of the constructs should not have any connection with COLD at all. When using it for COLD, the lay-out style (described in for example [10]) could just as well be added *after* instantiation of the expression by analysing the contents of the instantiation against these style rules.

3.1.3 Conclusion

We choose to adopt a lexical approach with "dumb" macro-constructs. This means that the language TEMPL will be a general purpose macro language and could be used for any other formal language in the way it is used now for COLD. We must however note that the specific lexical symbols we use in TEMPL are especially suited for COLD descriptions.

3.2 An example of TEMPL use

We will now give an example of how TEMPL can be used to describe a generic text and show how the instantiation can be obtained by using the instantiation language. This allows us to introduce the reader to the language in a direct way, and we can explain the notations and design issues of the language as they occur.

The following is an example of a TEMPL expression for $\texttt{TUP}_n$, a generalisation of the IGLOO components `TUP1`...`TUP25`. We will write the TEMPL expression in `typewriter` font.

3.2.1 The TEMPL expression

Before the actual description of the generic text, the parameters used have to be declared. We use several kinds of parameters : options, dimensions, singular string parameters and sequence string parameters.

We have chosen to explicitly declare the parameters. First of all, explicit declarations can avoid trivial errors. Second, declarations provide both the TEMPL writer and the specifier a clear overview of the parameters available. Lastly, it aids in checking the well-formedness of a TEMPL expression.

The declaration of options

```
OPTIONS
  unfold     {Help-text for unfold}     ;
  fold       {Help-text for fold}       ;
  Tup        {Help-text for Tup}        ;
  Items      {Help-text for Items}      ;
  proj       {Help-text for proj}       ;
  projinddef {Help-text for projinddef};
  projaxdef  {Help-text for projaxdef} ;
  tup        {Help-text for tup}        ;
```

This part declares the options that are used in the TEMPL expression. Options are two-valued, as booleans, but we say that an option is either *selected* or *not selected*. How options can be selected for a specific instantiation will be shown at the end of this section. By default all options are not selected.

For each option a (possible empty) help text is given. This help text can be read by the specifier, to aid him in the selection of options.

The options occupy a separate identifier name space from the other parameters. This allows overloading of identifier names which can improve readability of the TEMPL expressions.

Options are generally used to include certain pieces of text in the instantiation. To ensure that the specifier does not select in this way arbitrary pieces of text, *constraints* can be expressed over options, which define the sets of selected options that imply a meaningful instantiation.

The constraints on the options

```
CONSTRAINTS
  Items ;
  Tup ;
  unfold <=> NOT (fold) ;
  NOT (proj_inddef AND proj_axdef) ;
  (unfold AND proj) => (proj_inddef OR proj_axdef) ;
```

Constraints are specified on options with the intended meaning that if a constraint is not satisfied, then the instantiation should not take place. They are a means to provide the TEMPL developer with a certain level of control over the possible instantiations. For example, the above constraints state that options `Items` and `Tup` should always be selected; that `unfold` and `fold` exclude eachother; that `proj_inddef` should never be selected at the same time as `proj_axdef` and that `unfold` and `proj` imply that `proj_inddef` or `proj_axdef` should be selected.

The constraints use the ; as a low priority `AND`, as in COLD [5].

We discuss the dimensions and the parameters together, as they are closely linked.

The declaration of dimensions and parameters

```
DIMENSIONS
  n ;

PARAMETERS
  Items : Item[1,n] DETERMINES n {Help on Item} ;
  Tup   : Tup                    {Help on Tup}  ;
  proj  : proj[1,n]              {Help on proj} ;
  tup   : tup                    {Help on tup}  ;
```

The string parameters of the template must be preceded by an option. This option is called the *declaring* option and is introduced for the following reason.

More often than not, a piece of text in the actual TEMPL expression selected by an option uses substitutions for one or more string parameters. For example, the export signature of $\texttt{TUP}_n$ allows the function name referenced by `tup` to be included. By letting the option `tup` select this part of the export signature, giving a new value for `tup` has only effect if option `tup` is selected. Otherwise the new name for function `tup` is hidden (i.e. not exported) and the substitution did not have its intended effect.

We shall later see that by this way of combining options and string parameter declaration, the syntax for the instantiation language will become easier.

As mentioned, string parameters come in two flavours : sequence and singular. A *sequence* parameter is syntactically recognized by the *range indication* after its identifier, for example `[1,n]`. This means that 1 to n values can be given for this parameter. Each of these values can be referenced by appending an index to the string parameter identifier, e.g. `proj[2]`, `proj[i]`.

We see that the declared dimensions return as the upperbound for the ranges used in the sequence parameters. These dimensions are given a *derived* value. This derived value is determined by the number of values given for the determining parameter and the lowerbound for this parameter. This determining parameter is indicated by the keyword `DETERMINES`. If, for example, five values are given for `Item`, then the value for n will become 5.

Singular parameters can be given only one (single) value.

It is possible that the specifier does not provide values for some string parameter. In that case, when the value of the string parameter is needed, the (default) value is taken to be the literal text that is the string made of the identifier in the case of a singular parameter and the identifier appended by the string denotation of the integer index in the case of a sequence parameter. So, the identifier can play a double role : it is the denotation of the identifier for a parameter and it may be denoting the default value for that parameter.

If we adopt the convention that determing parameters must always receive values (this is the case in this example, because `Item[1,n]` is declared by option `Items`, which must be selected because it is mentioned in the constraints), then the sequence parameters that do not determine their dimension, can be given at most n values. These values should possibly extended with their default values if the number of values is less than the value of their dimension.

For each string parameter, a (possible empty) help text is given. This help text can be read by the specifier, to aid him in determining the meaning of the parameters.

The TEMPL body

The generic description follows immediately after our "dot"-symbol, `[]`. The instantiation will be based on this part of the TEMPL expression.

```
[]
COMPONENT [uc[Tup]] SPECIFICATION

EXPORT
   [Tup ? SORT Tup]
  [proj ? [i,1,n|FUNC proj[i] : Tup -> Item[i]|
          ]]
   [tup ? FUNC tup : [i,1,n|Item[i]| # ] -> Tup]

IMPORT
  [fold ? TUP[n][Items ?\[[i,1,n|Item[i]|,]\]]
          RENAMING  [proj ? [i,1,n|{proj}[i] TO proj[i]|
                            ]]
                    [tup ? {tup} TO tup ]
                    [Tup ? {Tup} TO Tup ]
          END
  ]
          [i,1,n|[uc[Item[i]]]|
          ]

CLASS
[unfold ?
    SORT Tup DEP [i,1,n|Item[i] |, ]

    DECL [i,1,n|i[i], j[i] : Item[i]|,],
         t : Tup

    FUNC tup : [i,1,n|Item[i]| # ] -> Tup
   [projinddef ?
    [i,1,n|
    FUNC proj[i] : Tup -> Item[i]
    IND  proj[i] ( tup ( [j,1,n|i[j]|, ] )) = i[i] |
    ]
   ]
```

```
[projaxdef ?
 [i,1,n|
 FUNC proj[i] : Tup -> Item[i]
 AXIOM proj[i] ( tup ( [j,1,n|i[j]|, ] )) = i[i] |
 ]
]
 AXIOM
      tup ([i,1,n|i[i]|, ])!;
      tup ([i,1,n|i[i]|, ]) = tup ([i,1,n|j[i]|, ]) =>
           ( [i,1,n|i[i] = j[i]| AND ] )

 PRED is_gen : Tup
 IND  is_gen ( tup ( [i,1,n|i[i]|, ] )))

 AXIOM is_gen (t)
]
```

Note that the most important TEMPL syntactic symbols are taken from the following set of characters : [,], |, {, }. In this actual TEMPL expression, a number of constructs is introduced. We will discuss them one by one.

The choice construct: [*condition*?*expr*] This is the TEMPL variant of the if-then construct used in many languages. Its informal meaning is that if *condition* holds, then *expr* is evaluated.

A variant of this construct is the if-then-else expression, denoted by [*condition*?*true-expr*|*false-expr*], with the informal meaning that if *condition* holds, then *true-expr* is evaluated, else *false-expr* is evaluated.

Both constructs allow for the optional inclusion of an expression.

The iteration construct: [*id*, *lwb*, *upb*|*expr*|*separation*] This is the TEMPL variant of the iteration expression, or for-loop that is present in many languages. The identifier *id* is a local identifier, its scope is limited to *expr*. The *lwb* and *upb* are simple arithmetic expressions build from integers and identifiers. The *separation* can be any expression, but may not contain a reference to the local identifier.

Its informal meaning is that *expr* is $upb - lwb + 1$ times evaluated, in which each evaluation *id* holds the value of the number of times that the expression is evaluated, increased with the lowerbound. These evaluations are separated by *separation*.

The indexing construct: [*index*] This construct is used in two ways:

- As an index in a sequence parameter
  Together with the preceding sequence parameter identifier, it selects the value of that parameter indicated by the integer value of *index*, which is a simple arithmetic expression.

- As a stand-alone construct
  It denotes the string that is made of the integer value of *index*.

Note that the difference between the two uses is that indexing requires the indexing construct to immediately follow the sequence parameter identifier.

Function calls: [*function-name*[*expr*]] For convenience a number of basic string functions is introduced. In this example we see the use of [uc[Tup]], which converts its evaluated expression argument to uppercase. The other functions are listed in the section on syntax, section 3.3.

The literal construct: *{literal}* This construct is used to indicate that a sequence of COLD-1 lexical units, *literal*, have to be evaluated literally. It can be viewed as a kind of escape sequence, and allows the use of TEMPL syntactic symbols in the source language and the use of parameter identifiers as ordinary strings.

The escape construct: *\character* This construct is used to indicate that the character following the \ is to be taken literally.

The substitution construct: *(invisible)* This construct substitutes the parameter by the value of that parameter. Note that in the expression no distinction is made between COLD symbols and TEMPL parameters. If a symbol is declared as a parameter, then it is used as a parameter and its place in the expression is substituted by its value. As one can see in the renaming section in the expression, this is not always wanted behaviour. So, when such a substitution should not take place, it is necessary to surround the symbol with { and }. We choose the string denotations for TEMPL-identifiers from the same set of characters that is used to denotate COLD-identifiers.

Invisible symbols like spaces and newlines These are used to separate textual elements, parameters, indexes etc or not, but also to indicate the format of the instantiated TEMPL expression. E.g. in the iteration construct, spaces and newlines in the separation part are recognized as part of that separation.

Implicit catenation All the above constructs evaluate to strings. They are implicitly catenated, so the complete expression is evaluated to one string.

This completes the TEMPL expression for $\texttt{TUP}_n$.

3.2.2 The language for the Arguments

Our use of options is similar to that of flags, that are typically used in the UNIX world. By this we mean that each option influences the outcome of the instantiation. Because by default no options are selected, we see that the argument language does not need to provide a means to deselect an option.

Giving values to the parameters implies the need to indicate for which parameter a sequence of values is meant for. It is our convention that each parameter p is declared by a unique option o and that giving values for p requires the selection of o. This would suggest that if parameter p is declared by option o, we could use o as a *keyword parameter* for the values of p.

As a conclusion, we propose the following input format for the generator:

`temples` *templ-name* { *-option* { *argument* } }

in which the $\{\cdots\}$-notation means zero or more repetitions of $\cdots$.

3.2.3 Some example instantiations

Example 1: A folded component

Suppose we call the generator as follows:

```
temples tup.templ -fold -Tup Desert -Items Fruit Dairy
                  -proj first last -tup mk_desert
```

The first argument indicates that the TEMPL expression in the file `tup.templ` is to be instantiated. Then, all the options that the specifier wants to select are given. The options that declare parameters can be postfixed with arguments for those parameters. An option is preceded by a `-`, so there is no ambiguity which argument is an option or which argument is a string-value. The options can be given in an arbitrary order, however the ordering of the string values is significant.

The resulting component

This specific call would lead to the following component:

```
COMPONENT DESERT SPECIFICATION

EXPORT
  SORT Desert
  FUNC first : Desert -> Fruit
  FUNC last  : Desert -> Dairy
  FUNC mk_desert : Fruit # Dairy -> Desert

IMPORT
  FRUIT,
  DAIRY,
  TUP2[Fruit, Dairy] RENAMING proj1 TO first
                            , proj2 TO last
                            , Tup2  TO Desert
                            , tup   TO mk_desert
                     END
END
```

With this example we address the discussion on *PME.Imports* again. If imported components were to be instantiated automatically by the system, this instantiation should have to be described in the TEMPL expression. However, because we choose to leave much freedom to the TEMPL developer, we cannot assume the presence of any built-in options in these imported TEMPL expressions. This means that generating for example the imported `TUP2` with as options `unfold` is in general not possible.

This is the reason that we propose that the system just retreives the names of the imported components and reports them to the specifier (see *PME.Imports*).

Example 2 : An unfolded component

If the call to the generator had been:

```
temples tup.templ -unfold -Tup Coordinate -Items Nat Nat
                  -tup coordinate
```

then the following would have been generated:

```
COMPONENT COORDINATE SPECIFICATION

EXPORT
  SORT Coordinate
  FUNC coordinate : Nat # Nat -> Coordinate

IMPORT
```

```
  NAT,
  NAT

CLASS

  SORT Coordinate DEP Nat
  FUNC coordinate : Nat # Nat -> Coordinate

  DECL i1, j1 : Nat, i2, j2 : Nat,
       t : Coordinate

  AXIOM coordinate(i1,i2)!;
        coordinate(i1,i2) = coordinate (j1, j2) =>
                            ( i1 = j1 AND i2 = j2 )

  PRED is_gen: Coordinate
  IND  is_gen(coordinate(i1,i2))

  AXIOM is_gen(t)

END
```

Note that in both the examples the constraints are satisfied.

3.3 Concrete Syntax of TEMPL

Now we have seen how TEMPL can be used to describe a generic TUP_n, it is time to show how in general a TEMPL expression can be written by giving the concrete syntax for TEMPL.

3.3.1 The BNF formalism used

We adopt the following notational convention for our concrete syntax, which is based on the Backus Nauer Form for denoting syntax:

- A non-terminal is written in small-caps font: NONTERMINAL
- A terminal is written in typewriter font: `terminal` and surrounded with quotes.
- We use the following meta-symbols:

$\alpha \mid \beta$	means choice of either α or β,
$[\alpha]$	means zero or one time α,
$\{\alpha\}$	means a sequence of zero or more α's,
`'a'...'z'`	means a selection of one character from the range `'a'` through `'z'`,
$\gamma ::= \delta$	is our notation for a production rule in the grammar,

  where α, β can be any sequence of terminals and nonterminals separated by spaces, δ is an expression built of (non)terminals and the first four constructs and γ is a single non-terminal.

3.3.2 The grammar

We start the grammar with the nonterminal TEMPL:

TEMPL ::= PARAMETERPRELUDE '□' EXPRESSIONS

```
ParameterPrelude ::= [ OptionDeclaration ]
                     [ ConstraintSection ]
                     [ DimensionDeclaration ]
                     [ ParameterDeclaration ]

OptionDeclaration ::= 'OPTIONS' Options

Options ::= Option [ ';' Options ]

Option ::= Identifier Help

Help ::= '{' AnyChars '}'

ConstraintSection ::= 'CONSTRAINTS' Constraints

Constraints ::= Constraint [ ';' Constraints ]

Constraint ::= Constraint 'AND' Constraint
             | Constraint 'OR' Constraint
             | Constraint '<=>' Constraint
             | Constraint '=>' Constraint
             | 'NOT' Constraint
             | '(' Constraint ')'
             | Identifier

DimensionDeclaration ::= 'DIMENSIONS' Dimensions

Dimensions ::= Dimension [ ';' Dimensions ]

Dimension ::= Identifier

ParameterDeclaration ::= 'PARAMETERS' Parameters

Parameters ::= Parameter [ ';' Parameters ]

Parameter ::= SingleParameter | SequenceParameter

SingleParameter ::= Identifier ':' Identifier Help

SequenceParameter ::= Identifier ':' Identifier '[' Integer ','
                      Identifier ']' [ DeterminesSection ] Help

DeterminesSection ::= 'DETERMINES' Dimension

Expressions ::= { Expression }

Expression ::= SequenceExpression
             | IfThenExpression
             | IfThenElseExpression
             | SubstitutionExpression
             | LiteralExpression
             | TextExpression
             | ValueExpression
             | FormatExpression
             | FunctionCall
```

SEQUENCEEXPRESSION ::= '[' IDENTIFIER ',' MATHEXPRS ',' MATHEXPRS
'|' EXPRESSIONS '|' SEPARATOR ']'

IFTHENEXPRESSION ::= '[' CONSTRAINT '?' EXPRESSIONS ']'

IFTHENELSEEXPRESSION ::= '[' CONSTRAINT '?' EXPRESSIONS
'|' EXPRESSIONS ']'

SUBSTITUTIONEXPRESSION ::= IDENTIFIER ['[' MATHEXPRS ']']

LITERALEXPRESSION ::= '{' ANYCHARS '}'

TEXTEXPRESSION ::= COLDCHARS

VALUEEXPRESSION ::= '[' MATHEXPRS ']'

FUNCTIONCALL ::= '[' IDENTIFIER '[' ARGEXPRESSIONS ']]'

ARGEXPRESSIONS ::= EXPRESSIONS '|' ARGEXPRESSIONS
EXPRESSIONS

MATHEXPRS ::= '+' MATHEXPR | '-' MATHEXPR | MATHEXPR

MATHEXPR ::= MATHEXPR '+' MATHEXPR
| MATHEXPR '-' MATHEXPR
| '(' MATHEXPRS ')'
| IDENTIFIER
| INTEGER

SEPARATOR ::= EXPRESSIONS

3.3.3 Lexical conventions

In the above grammar, spaces and newlines are significant where EXPRESSIONS, EXPRESSION or HELP can be produced. This is indicated by the following production rule:

FORMATEXPRESSION ::= { SP | NL }

where SP stands for space and NL stands for newline. At the other places these characters are ignored and are used there to separate symbols.

The syntactic class of identifiers is built from the following regular expression:

IDENTIFIER ∈ ('A'...'Z' + 'a'...'z') ('A'...'Z' + 'a'...'z' + '0'...'9' + '_')*

The syntactic class of integers is built from the following regular expression:

INTEGER ∈ ('0'...'9')+

The syntactic class of ANYCHARS is built from the following regular expression:

ANYCHARS ∈ (NUL...'z' + '|' + '~'...DEL)+

where we note that it is not possible to use a simple escape sequence in the production HELP.

The syntactic class of COLDCHARS includes all characters that are not identifiers and which are not TEMPL syntactic symbols:

COLDCHARS ∈ ('!'...'@' + '^'...'`' + '~')+

The simple escape sequence \n, n any character, can be used anywhere in TEXTEXPRESSION or LITERALEXPRESSION. Note that the use of \ as an escape indicator requires that one should write \\ whenever \ is to be generated.

3.3.4 Remarks on the concrete syntax

Because spaces (SP) and newlines (NL) derived from EXPRESSIONS have a formatting meaning, they are mentioned here as concrete syntactic entities. However, they have no meaning in the *control* parts of the TEMPL expression. By the control parts we mean that part of the expression which is evaluated by TEMPL and does not return in the instantiated text, e.g. the '[' IDENT ',' MATHEXPRS ',' MATHEXPRS '|' part of a SEQUENCEEXPRESSION can be arbitrarily interleaved with spaces and newlines without any effect on the format of the instantiated text. It is even so that the space occupied by the characters in the control part of a TEMPL construct is left out in the instantiation. However, spaces and newlines in the SEPARATOR part of the SEQUENCEEXPRESSION do have effect. Also spaces and newlines in the declaration part of the TEMPL expression have no meaning but separation of symbols.

Note that ANYCHARS includes all characters, *except* '{' and '}'. There is another escape construct, '\' followed by any character. The backslash indicates that the next character is to be taken literally. Although this escape sequence is only really needed for the characters { and }, we have chosen to extend this facility to any other character.

The non-terminal EXPRESSION is the most important in this definition. Basically, the actual expression of a TEMPL expression is just an arbitrary sequence of EXPRESSIONS. This is possible because the non-terminals FORMATEXPRESSION and TEXTEXPRESSION can hold sequences of arbitrary length of COLDCHARS and SP, NL.

Please note that, although for example 'OPTIONS' is a TEMPL keyword, it can be used freely in the TEMPL expression *after* the symbol []. The same goes for any of the TEMPL declaration keywords.

3.3.5 Implemented string functions

The follwing basic string functions have been implemented:

- [uc[a]], for uppercase conversion of the *evaluation* of a;
- [lc[a]], for lowercase conversion of the *evaluation* of a;
- [tk[a|n]], to insert the string made of the first n+1 characters of the *evaluation* of a;
- [dp[a|n]], to insert the string made of the *evaluation* of a with the first n+1 characters removed;
- [ln[a]], which results in the string representation of the length of the *evaluation* of a.

The a in the above stands for an arbitrary EXPRESSIONS, the n should either be a string representation of an integer or evaluate somehow to a string representation of an integer, as in for example a VALUEEXPRESSION. All the above fall under the nonterminal EXPRESSION, which means that they all evaluate to a string. This is important to note, because it is not possible to use for example [ln[a]] as an upperbound to an iteration expression. However, because [ln[a]] evaluates to a string representation of an integer, it can be used as the n argument to for example [dp[a|[ln[a]]]][2].

The following relation can be given for the dp, tk and ln: If i is any value between 0 and ln[a] then

$$[tk[a|[i]]][dp[a|[i]]] \equiv a$$

[2] which yields ε

3.4 Semantics of TEMPL

In this section we will give the denotational semantics of TEMPL expressions. After we have clarified the concept of denotational semantics we will give the semantics in parts. First, we give the semantics of the *static* constructs of the TEMPL expression, which is also a good introduction to the notation. Then we will give the semantics of the *iteration* expression, which introduces a recursive definition, and the *function calls*. After we have taken the declarations and the arguments into account, we will see that instantiation can be described using a suitable combination of the functions defined in this section.

3.4.1 The meaning of denotational semantics

Take an arbitrary TEMPL expression. We may ask ourselves: What is the meaning of this expression combined with these arguments?

Denotational semantics allows us to give this meaning, by describing essentially a *mapping* from all the possible expressions of a language to their meaning, also called their *denotation*.

The advantages of describing the denotational semantics are numerous. First, all constructs in the language are assigned a meaning. This allows a user of the language to see precisely what the meaning of a certain construct is. Furthermore, by giving the meanings of the basic constructs, we hope to achieve that any expression built from these basic constructs can be assigned a meaning by combining the meanings of the encoutered basic constructs. This concept of *compositionality* will allow us to describe instantiation at a very abstract level. The style of our semantic descriptions allows us to change some minor aspect in the language and see what effect it has on the meaning of the resulting language definition. Finally, we can propose extensions to the language by giving their valuation functions without having to implement them.

The way we notate the denotational semantics is taken from Schmidt, which is treated extensively in [16].

3.4.2 Notation used

We assume that the reader is familiar with the concept of sets and the λ-notation for functions. We introduce the following notations for operations on sets:

Product of sets

$R \times S = \{(x, y) \mid x \in R \text{ and } y \in S\}$
with operations:
$(x, y)\downarrow_1 = x$
$(x, y)\downarrow_2 = y$

Disjoint sum of sets

$R + S = \{(zero, x) \mid x \in R\} \cup \{(one, y) \mid y \in S\}$
with operations:
for $x \in R$, $\text{in}R(x) = (zero, x)$
for $y \in S$, $\text{in}S(y) = (one, y)$

The above is easily generalized to n-products and n-disjoint sums of sets.

Functions on sets

$f : A \to B$
We use the following function updating expression.
Let $f : A \to B$. We let $[a_0 \mapsto b_0]f$ be the function that behaves like f, except that it maps the element $a_0 \in A$ to $b_0 \in B$, that is:
$([a_0 \mapsto b_0]f)(a_0) = b_0$
$([a_0 \mapsto b_0]f)(a) = f(a)$ for all $a \in A$ s.t. $a \neq a_0$

Non-termination

$\perp$
We write $\perp$ for non-termination, or "no value at all". Any set A can be extended with a fresh element $\perp_A$ to give the *lifted* domain $A_\perp$.
$A \cup \{\perp_A\} = A_\perp$

Strictness

A function $f : A_\perp \to B_\perp$ that is undefined at a has the property that $f(a) = \perp$. We write $f = \underline{\lambda} x.\alpha$ to denote the strict mapping:
$f(\perp) = \perp$
$f(a) = [a/x]\alpha$
We use the following abbreviation:
(let $x = e_1$ **in** e_2**)** for $(\underline{\lambda} x.e_2)e_1$
In this section we assume all functions to be strict, unless stated otherwise.

Identity

We define the function $Id : A \to A$ on any domain A to be that function s.t. if $a \in A$ then $Id(a) = a$.

3.4.3 Semantic Algebras

We use the notation to give the *semantic algebras*, which define the domains that are used as the denotations of the expressions in the language and the operations on those domains.

Primitive domains

We will first define a number of *primitive* domains, which form the basis on which we can build our compound domains later on.

Integer numbers Domain **Integer** = $\mathbb{Z}$
Operations:
..., -2, -1, 0, 1, 2, ... : **Integer**
$\cdot + \cdot$: **Integer** × **Integer** → **Integer**
$\cdot - \cdot$: **Integer** × **Integer** → **Integer**
$-\cdot$: **Integer** → **Integer**
$\cdot > \cdot$: **Integer** × **Integer**→ **Boolean**
$\cdot < \cdot$: **Integer** × **Integer**→ **Boolean**
$\cdot = \cdot$: **Integer** × **Integer**→ **Boolean**
$10^{\cdot}$: **Integer** → **Integer**
We take as numbers the domain of **Integer**. Because intermediate results may be negative, we

cannot use the natural numbers. Besides that, implementation poses a finite limit on the numbers that we can use. The operations mentioned have their usual meaning. We have defined them in a convenient infix notation. The exponentation function $10^{\cdot}$ is used in the string conversion function, which definition follows later.

Truth values Domain **Boolean** = **B**
Operations:
true, false : **Boolean**

$\cdot \wedge \cdot$	:	**Boolean** × **Boolean** → **Boolean**
$\cdot \vee \cdot$	:	**Boolean** × **Boolean** → **Boolean**
$\cdot \Leftrightarrow \cdot$	:	**Boolean** × **Boolean** → **Boolean**
$\cdot \Rightarrow \cdot$	:	**Boolean** × **Boolean** → **Boolean**
$\neg \cdot$	:	**Boolean** → **Boolean**
(if $\cdot$ **then** $\cdot$ **else** $\cdot$**)**	:	**Boolean** × **A** × **A** → **A**

We use booleans to model the truth values of our constraints. The operations are again written in infix notation. The last function is also called the *conditional*. This conditional takes three arguments: a **Boolean** and two elements of the arbitrary domain **A**. For elements $a, b \in \mathbf{A}$, it is defined as:

(if true **then** a **else** b**)** $= a$
(if false **then** a **else** b**)** $= b$

The conditional is non-strict in its second and third argument, so **(if** true **then** a **else** $\perp$**)** $= a$.

Characters Domain **Char**
Operations:
0, ..., *9*, *a*, *b*, ..., *Z*, *&*, *!* etc. : **Char**
Ctoi : **Char** → **Integer**$_\perp$

Ctoi(*0*)	=	0
Ctoi(*1*)	=	1
Ctoi(*2*)	=	2
⋮		⋮
Ctoi(*9*)	=	9
Ctoi(*c*)	=	$\perp$, otherwise

Ctoi maps a character to an integer. Only the characters *0*, ...,*9* are mapped to a non-$\perp$ value.
ItoC : **Integer** → **Char**$_\perp$

ItoC(0)	=	*0*
ItoC(1)	=	*1*
ItoC(2)	=	*2*
⋮		⋮
ItoC(9)	=	*9*
ItoC(*n*)	=	$\perp$, otherwise

ItoC maps an integer to its character representation. Only the integers between 0,...,9 are mapped to a non-$\perp$ value.

Strings Domain **String**
Operations:
ϵ : **String**
$\cdot : \cdot$: **Char** × **String** → **String**
$\cdot ::: \cdot$: **String** × **String** → **String**
$\cdot = \cdot$: **String** × **String** → **Boolean**
ln$(\cdot)$: **String** → **Integer**
uc$(\cdot)$: **String** → **String**

$lc(\cdot)$: **String** $\rightarrow$ **String**
The result of the evaluation of the expression is a string. Strings can be constructed by means of the : (catenation) operation. We write *abc* as a shorthand for $a{:}(b{:}(c{:}\epsilon))$. **String** equality $(\cdot{=}\cdot)$ and **String** length ($ln(\cdot)$) are defined in the conventional way.
$hd(\cdot)$: **String** $\rightarrow$ **Char**$_\perp$
$hd(\epsilon) = \perp$
$hd(c : s) = c$
$tl(\cdot)$: **String** $\rightarrow$ **String**$_\perp$
$tl(\epsilon) = \perp$
$tl(c : s) = s$

The following functions are needed for the TEMPL functions that operate on strings:

tk : **String**$_\perp$ $\rightarrow$ **Integer**$_\perp$ $\rightarrow$ **String**$_\perp$
tk = $\underline{\lambda}s.\underline{\lambda}i.$**if** $i <= 0$
then ϵ
else if $s = \epsilon$
then ϵ
else $hd(s) : tk(s)(i-1)$

$tk(s)(n)$ (*take*) maps a string s to the substring of s consisting of the first n characters of s.

dp : **String**$_\perp$ $\rightarrow$ **Integer**$_\perp$ $\rightarrow$ **String**$_\perp$
dp = $\underline{\lambda}s.\underline{\lambda}i.$**if** $i <= 0$
then s
else if $s = \epsilon$
then ϵ
else $dp(tl(s))(i-1)$

$dp(s)(n)$ (*drop*) maps a string s to the substring of s with the first n characters of s removed.

The functions *uc* and *lc* map the characters of a string to uppercase and lowercase characters respectively.

Str : **Integer**$_\perp$ $\rightarrow$ **String**$_\perp$
Str = $\underline{\lambda}i.$**if** $i > 9$
then $Str(i$ **div** $10) :: Str(i$ **mod** $10)$
else if $i < 0$
then $\perp$
else $(ItoC(i)){:}\epsilon$

Str converts an integer to its string representation. We disallow conversion of negative integers by mapping them to $\perp$.

I : **String**$_\perp$ $\rightarrow$ **Integer**$_\perp$
I = $\underline{\lambda}s.$**if** $s = \epsilon$
then 0
else $(CtoI(hd(s))) \times 10^{ln(tl(s))} + I(tl(s))$

I converts the string representation of an integer to its integer value.

Sequences of strings Domain **StrSeq**
Operations:
ϵ : **StrSeq**
hd : **StrSeq** $\rightarrow$ **String**
tl : **StrSeq** $\rightarrow$ **StrSeq**
ln : **StrSeq** $\rightarrow$ **Integer**
$ln = \lambda s.$**if** $s = \epsilon$ **then** 0 **else** $1 + ln(tl(s))$
$cons$: **String** $\rightarrow$ **StrSeq** $\rightarrow$ **StrSeq**
$access$: **Integer** $\rightarrow$ **StrSeq** $\rightarrow$ **String**
$access = \lambda i.$**if** $i <= 0$ **then** hd **else** $access(i-1)(tl)$
$append$: **StrSeq** $\rightarrow$ **Integer** $\rightarrow$ **Integer** $\rightarrow$ **String** $\rightarrow$ **StrSeq**

$append = \lambda s.\lambda i.\lambda j.\lambda v.$
if $s = \epsilon$
then if $i - j > 0$
then ϵ
else $cons\ (v::(\mathbf{Str}(i)))(append(s)(i+1)(j)(v))$
else $cons(hd(s))(append(tl(s))(i+1)(j)(v))$

Most functions will look familiar to the reader. *hd(s)* returns the first **String** of s, while *tl(s)* returns s without the first element. *cons(s)(sq)* returns a new **StrSeq**, with as its head the argument s and as its tail the argument sq. The following property holds for these operations:

$$s \neq \epsilon \Rightarrow cons(hd(s))(tl(s)) = s$$

The function *access(n)(s)* allows access to an arbitrary element of a **StrSeq**. It is defined recursively. The function *append(s)(i)(j)(v)* is somewhat more interesting: its purpose is to extend s with default values. These default values consist of v followed by an index number in the range $i, \ldots, j$. If s is ϵ, then $j - i$ values will be appended to the sequence. For example, *append (cons aap (cons mies (cons noot ϵ)))(1)(5)(values)* will result in *(cons aap (cons (mies (cons noot (cons (values4 (cons values5 ϵ)))))))*

Identifiers Domain: **Ide**
Operations:
Ide → String
We just state that there are identifiers and that there is a nameless function which maps an identifier to its string denotation.

Compound domains

We now can define domains that are built from these primitive domains. Because we allow overloading of our identifiers, we propose four stores which map these identifiers to their values. We shall see that the following four domains have many operations in common, but for completeness we will give the definitions of all these functions.

Options Domain: **Option = Ide → (String × Boolean)$_\perp$**
Operations:
empty : **Option**
$empty = \lambda i.\perp$
new : **Option → Ide → (String × Boolean) → Option**
$new = \lambda o.\lambda i.\lambda(h,b).[i \mapsto (h,b)]o$
select : **Option → Ide → Option**
$select = \lambda o.\lambda i.$ **let** $(h,b) = o(i)$ **in** $[i \mapsto (h, true)]o$
access : **Option → Ide → Boolean$_\perp$**
$access = \lambda o.\lambda i.$ **let** $(h,b) = o(i)$ **in** b
empty is a store of options in which all identifiers are mapped onto $\perp$. *new(o)(i)((h,b))* modifies a store o of options by modifying the value for i to (h,b). *select(o)(i)* modifies the **Boolean** field of i in o and returns this new o. *access(o)(i)* returns the value stored in the **Boolean** field.

Singular Parameters Domain: **Singular = Ide → (String × Ide × String)$_\perp$**
Operations:
empty : **Singular**
$empty = \lambda i.\perp$
new : **Singular → Ide → (String × Ide × String) → Singular**

$new = \lambda si.\lambda i.\lambda(h,o,v).[i \mapsto (h,o,v)]si$
option : **Singular** $\rightarrow$ **Ide** $\rightarrow$ **Ide**$_\perp$
$option = \lambda si.\lambda i.$ **let** $(h,o,v) = si(i)$ **in** o
value : **Singular** $\rightarrow$ **Ide** $\rightarrow$ **String**$_\perp$
$value = \lambda si.\lambda i.$ **let** $(h,o,v) = si(i)$ **in** v
setvalue : **Singular** $\rightarrow$ **Ide** $\rightarrow$ **String** $\rightarrow$ **Singular**
$setvalue = \lambda si.\lambda i.\lambda v.$ **let** $(h,o,v') = si(i)$ **in** $[i \mapsto (h,o,v)]si$
access : **Singular** $\rightarrow$ **Ide** $\rightarrow$ **String**$_\perp$
$access = \lambda si.\lambda i.$ **let** $(h,o,v) = si(i)$ **in** v
The domain of the singular parameters has similar operations to the domain of options. *setvalue(si)(i)(v)* modifies the store *si* by modifying the result of the map on i to v.

Sequence Parameters Domain: **Sequence** = **Ide** $\rightarrow$ (**String** $\times$ **Ide** $\times$ **Integer** $\times$ **Ide** $\times$ **Boolean** $\times$ **StrSeq**)$_\perp$
Operations:
empty : **Sequence**
$empty = \lambda i.\perp$
new : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ (**String** $\times$ **Ide** $\times$ **Integer** $\times$ **Ide** $\times$ **Boolean** $\times$ **StrSeq**) $\rightarrow$ **Sequence**
$new = \lambda se.\lambda i.\lambda(h,o,l,u,b,v).[i \mapsto (h,o,l,u,b,v)]se$
option : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **Ide**$_\perp$
$option = \lambda se.\lambda i.$ **let** $(h,o,l,u,b,v) = se(i)$ **in** o
lowerbound : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **Integer**$_\perp$
$lowerbound = \lambda se.\lambda i.$ **let** $(h,o,l,u,b,v) = se(i)$ **in** l
upperbound : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **Ide**$_\perp$
$upperbound = \lambda se.\lambda i.$ **let** $(h,o,l,u,b,v) = se(i)$ **in** u
determines : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **Boolean**$_\perp$
$determines = \lambda se.\lambda i.$ **let** $(h,o,l,u,b,v) = se(i)$ **in** b
values : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **StrSeq**$_\perp$
$values = \lambda se.\lambda i.$ **let** $(h,o,l,u,b,v) = se(i)$ **in** v
setvalues : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **StrSeq** $\rightarrow$ **Sequence**
$setvalues = \lambda se.\lambda i.\lambda v.$ **let** $(h,o,l,u,b,v') = se(i)$ **in** $[i \mapsto (h,o,l,u,b,v)]se$

access : **Sequence** $\rightarrow$ **Ide** $\rightarrow$ **Dimension** $\rightarrow$ **Integer** $\rightarrow$ **String**$_\perp$
$access = \lambda se.\lambda i.\lambda d.\lambda n.$
let $(h,o,l,u,b,v) = se(i)$ **in**
if $(n < l) \vee (n > access(d)(u))$
then $\perp$
else $access(v)(n-l)$

As for the singular parameter domain, sequence parameters have similar operations to the domain of options. The non-trivial function here is *access(se)(i)(d)(n)* which is a kind of indexing operation. Our access function on *StrSeq* is defined from 0, so we must substract the lowerbound l from the index n, in order to find the wanted string in the values of i. We must also make sure that the index n is in range, if not, we should return $\perp$.

Dimensions Domain: **Dimension** = **Ide** $\rightarrow$ **Integer**$_\perp$
Operations:
new : **Dimension** $\rightarrow$ **Ide** $\rightarrow$ **Integer** $\rightarrow$ **Dimension**
$new = \lambda d.\lambda i.\lambda n.[i \mapsto n]d$
empty : **Dimension**
$empty = \lambda i.\perp$

Table 3.1: Abstract syntax of simple TEMPL constructs

StrExpr			MathExpr			Constraint		
S	::=	C ? S	M	::=	M + M	C	::=	C and C
	|	C ? S | S		|	M - M		|	C or C
	|	P [M]		|	- M		|	not C
	|	P		|	+ M		|	C <=> C
	|	[M]		|	I		|	C => C
	|	S S		|	N		|	I
	|	T		.			.	
	|	ϵ						
	.							

where
P, I $\in$ **Ide**,
T, N $\in$ **String**

setvalue : **Dimension** $\rightarrow$ **Ide** $\rightarrow$ **Integer** $\rightarrow$ **Dimension**
$setvalue = \lambda d.\lambda i.\lambda n.[i \mapsto n]d$
access : **Dimension** $\rightarrow$ **Ide** $\rightarrow$ **Integer**$_\perp$
$access = \lambda d.\underline{\lambda} i.d(i)$
The store of dimensions is a simple map from identifiers to their integer value.

Store of Identifiers Domain σ : **Store** = **Option** $\times$ **Singular** $\times$ **Sequence** $\times$ **Dimension**
Operations:
empty : **Store**
$empty = \lambda s.(empty, empty, empty, empty)$
O : **Store** $\rightarrow$ **Option**
$O = \lambda s.s\downarrow_1$
Si : **Store** $\rightarrow$ **Singular**
$Si = \lambda s.s\downarrow_2$
Se : **Store** $\rightarrow$ **Sequence**
$Se = \lambda s.s\downarrow_3$
D : **Store** $\rightarrow$ **Dimension**
$D = \lambda s.s\downarrow_4$
The store used in the actual semantic definitions is a convenient tuple of the optionstore, the store of singular parameters, the store of sequence parameters and the store of dimensions. The above described operations are meant for convenience.

3.4.4 Semantics of the basic constructs

In the semantics of TEMPL we want to focus on the relevant aspects of the expressions in the language. So, we use an *abstract* version of the concrete syntax mentioned in section 3.3. We give the abstract syntax for the basic constructs in table 3.1. In this table we notate syntactic entities as small caps single letters. For convenient reading, some extra symbols have been added to distinguish the productions.

The following valuation functions describe how these syntactic entities are mapped onto the semantic algebras. For now we do not give a semantic definition for the iteration construct. We also assume that the store σ has been filled with values for the encountered identifiers.

In table 3.2 we give the signature of the valuation functions that we are going to define next.

Table 3.2: Signature of the valuation functions

S : StrExpr→ **Store** → **String**$_\perp$
M : MathExpr→ **Store** → **Integer**$_\perp$
C : Constraint→ **Store** → **Boolean**$_\perp$

S : StrExpr→ **Store** → **String**$_\perp$		
S⟦ s s′ ⟧σ	=	**S**⟦ s ⟧σ :: **S**⟦ s′ ⟧σ
S⟦ c ? s ⟧σ	=	**S**⟦ c ? s | ϵ ⟧σ
S⟦ c ? s | s′ ⟧σ	=	**if C**⟦ c ⟧σ **then S**⟦ s ⟧σ **else S**⟦ s′ ⟧σ
S⟦ p ⟧σ	=	$access(Si(\sigma))$(p)
S⟦ p [m] ⟧σ	=	$access(Se(\sigma))$(p) $(D(\sigma))$(**M**⟦ m ⟧σ)
S⟦ [m] ⟧σ	=	Str(**M**⟦ m ⟧σ)
S⟦ t ⟧σ	=	t
S⟦ ϵ ⟧σ	=	ϵ

Let us examine the details of this function. In a TEMPL expression, the store σ is not modified by any expression. That is why we can give the store as an argument to both **S**⟦ s ⟧ and **S**⟦ s′ ⟧. We define **S** on the if-then-expression as a special case of the if-then-else-expression, by letting the else expression equal ϵ.

Singular parameter substitution is defined by accessing and returning the value of that specific parameter in the singular parameter part of the store. Sequence parameter substitution is defined as the accessing of the value of the parameter at an index, which is found by valuating m.

An arbitrary string t has as its denotation itself. We have distinguished the case where this string is ϵ, for reason of clarity.

M : MathExpr→ **Store** → **Integer**$_\perp$		
M⟦ m + m′ ⟧σ	=	**M**⟦ m ⟧σ + **M**⟦ m′ ⟧σ
M⟦ m − m′ ⟧σ	=	**M**⟦ m ⟧σ − **M**⟦ m′ ⟧σ
M⟦ + m ⟧σ	=	**M**⟦ m ⟧σ
M⟦ − m ⟧σ	=	− **M**⟦ m ⟧σ
M⟦ i ⟧σ	=	$access(D(\sigma))$(i)
M⟦ n ⟧σ	=	I(n)

M returns the integer value of a MathExpr and needs the store because possibly a dimension identifier is used in the MathExpr.

C : Constraint→ **Store** → **Boolean**$_\perp$		
C⟦ c and c′ ⟧σ	=	**C**⟦ c ⟧σ ∧ **C**⟦ c′ ⟧σ
C⟦ c or c′ ⟧σ	=	**C**⟦ c ⟧σ ∨ **C**⟦ c′ ⟧σ
C⟦ c <=> c′ ⟧σ	=	**C**⟦ c ⟧σ ⇔ **C**⟦ c′ ⟧σ
C⟦ c => c′ ⟧σ	=	**C**⟦ c ⟧σ ⇒ **C**⟦ c′ ⟧σ
C⟦ not c ⟧σ	=	¬**C**⟦ c ⟧σ
C⟦ i ⟧σ	=	$access(O(\sigma))$(i)

C is a function that maps a Constraint to a **boolean**, using $O(\sigma)$ as the store for its option identifiers.

Table 3.3: Full abstract syntax of TEMPL constructs

StrExpr		
S	::=	C ? S
	|	C ? S | S
	|	I , M , M | S | T
	|	[F[FA]]
	|	[F[FA | FA]]
	|	P [M]
	|	P
	|	[M]
	|	S S
	|	T
	|	ϵ
	.	

where

FA $\in$ StrExpr,
I, P, F $\in$ **Ide**,
T $\in$ **String**

3.4.5 Semantics of the sequence expression and function calls

Our use of local identifiers allows us to easily introduce the valuation of the sequence expression. Because the scope of a local identifier i is limited to the body of its sequence expression, we can adjust the value of i to the current number that the body is evaluated. This is defined in the function **It**.

Function calls are evaluated by first evaluating their arguments and then applying the function to it. The evaluations are strings, so the function should be defined on a **StrSeq**. The adapted syntax for StrExpr is listed in table 3.3.

S :StrExpr$\rightarrow$ **Store** $\rightarrow$ **String**$_\perp$

S[[I , M , M′ | S | T]]σ =		
		It(I, **M**[[M]]σ, **M**[[M′]]σ, S, **S**[[T]]σ,σ)
S[[S S′]]σ	=	**S**[[S]]σ :: **S**[[S′]]σ
S[[C ? S]]σ	=	**S**[[C ? S | ϵ]]σ
S[[C ? S | S′]]σ	=	**if C**[[C]]σ **then S**[[S]]σ **else S**[[S′]]σ)
S[[[F [FA]]]]σ	=	**F**[[F [FA]]]σ
S[[[F [FA | FA′]]]]σ	=	**F**[[F[FA | FA′]]]σ
S[[P]]σ	=	$access(Si(\sigma))$(P)
S[[P [M]]]σ	=	$access(Se(\sigma))$(P) $(D(\sigma))$(**M**[[M]]σ)
S[[[M]]]σ	=	**Str**(**M**[[M]]σ)
S[[T]]σ	=	T
S[[ϵ]]σ	=	ϵ

It : **Ide** $\times$ **Integer** $\times$ **Integer**$\times$ StrExpr$\times$**String** $\times$ **Store** $\rightarrow$ **String**$_\perp$

It(I, l, u, S, t, σ)	=	**if** $l > u$
		then ϵ
		else
		if $l = u$
		then S[[S]]($setvalue$(I)(l)($D(\sigma)$))
		else S[[S]]($setvalue$(I)(l)($D(\sigma)$)) :: t :: **It**(I, $l+1$, u, S, t, σ)

Table 3.4: Abstract syntax of TEMPL declarations

DECLARATIONS
DS ::= OD; DD; PD

OPTIONDECL	DIMENSIONDECL	PARAMETERDECL
OD ::= O; OD	DD ::= D; DD	PD ::= AP; PD
| ϵ	| ϵ	| ϵ
O ::= I H	D ::= I	AP ::= I : P H
		| I : P [L,U] H
		| I : P [L,U] det U H

where

O $\in$ OPTION,
P, I, U $\in$ **Ide**,
H $\in$ **String**

The **It** function is recursively defined. If the lowerbound l exceeds the upperbound u then it returns ϵ. Otherwise, if the lowerbound l is equal to the upperbound u then it returns the evaluation of the body S, with an adjusted store in which the local identifier I has the value of l. If l is smaller than u, then the local identifier is assigned the value of the lowerbound l in the evaluation of S, and this is catenated to the evaluation of the separator (for which the old store is an argument!) and the recursive value of **It**, in which l is increased by one.

Because **It** is defined in terms of itself, we need to show that this definition is indeed correct. By strictness of $<$ and the choice function we have that if one of the values l, u should equal $\perp$ then the result of **It** will also equal $\perp$. Assuming $\mathbf{S}[\![$ S $]\!]\sigma$ and $\mathbf{S}[\![$ T $]\!]\sigma$ are finite, we have to see whether the use of the **It** function is finite. We see that this function has as an argument $l+1$ and the same u, which implies that at some time in the recursive unfolding of **It** the condition $l > u$ will hold.

F :STREXPR$\rightarrow$ **Store** $\rightarrow$ **String**$_\perp$
$\mathbf{F}[\![$ uc[FA] $]\!]\sigma = uc(\mathbf{S}[\![$ FA $]\!]\sigma)$
$\mathbf{F}[\![$ lc[FA] $]\!]\sigma = lc(\mathbf{S}[\![$ FA $]\!]\sigma)$
$\mathbf{F}[\![$ tk[FA | FA'] $]\!]\sigma = tk(\mathbf{S}[\![$ FA $]\!]\sigma)(\mathbf{I}(\mathbf{S}[\![$ FA' $]\!]\sigma))$
$\mathbf{F}[\![$ dp[FA | FA'] $]\!]\sigma = dp(\mathbf{S}[\![$ FA $]\!]\sigma)(\mathbf{I}(\mathbf{S}[\![$ FA' $]\!]\sigma))$
$\mathbf{F}[\![$ ln[FA] $]\!]\sigma = Str(ln(\mathbf{S}[\![$ FA $]\!]\sigma))$

The valuation function of a function call is defined by looking at the possible function identifiers and taking corresponding action on the argument list. Only those arguments that are necessary are evaluated. It is allowed to append more arguments, but these arguments are ignored in the evaluation.

3.4.6 Semantics of the declarations

Previously, we assumed that the store of global identifiers was a static element. By defining the semantics of the declaration part of the TEMPL syntax we see how this store is build. Note that we do not yet give the semantics nor the abstract syntax of the constraints section; the constraints section is not dependent on its current position in the TEMPL expression, it could be placed anywhere after the option declarations and before the actual TEMPL expression. We will discuss it later on.

Table 3.5: Abstract syntax of the constraints section

CONSTRAINTS		
CS	::=	C ; CS
	|	ϵ

The valuation function **DS** just states that parameter declarations are processed after dimension declarations which are processed after option declarations.

DS: DECLARATIONS → **Store** → **Store**		
DS⟦ OD; DD; PD ⟧σ	=	**PD**⟦ PD ⟧(**DD**⟦ DD ⟧(**OD**⟦ OD ⟧σ))

The valuation functions for the declaration part of a TEMPL expression use a **Store** argument to create a **Store**.

OD: OPTIONDECL → **Store** → **Store**		
OD⟦ O; OD ⟧σ	=	**OD**⟦ OD ⟧(**O**⟦ O ⟧σ)
OD⟦ ϵ ⟧σ	=	$Id(\sigma)$

O: OPTION → **Store** → **Store**		
O⟦ I H ⟧σ	=	**let** $(o, si, se, d) = \sigma$ **in** $(new(o)(\text{I})(\text{H}, \text{false}), si, se, d)$

The initial boolean value of an option is false, or equivalently, not selected.

DD: DIMENSIONDECL → **Store** → **Store**		
DD⟦ D; DD ⟧σ	=	**DD**⟦ DD ⟧(**D**⟦ D ⟧σ)
DD⟦ ϵ ⟧σ	=	$Id(\sigma)$

D: DIMENSION → **Store** → **Store**		
D⟦ I ⟧σ	=	**let** $(o, si, se, d) = \sigma$ **in** $(o, si, se, new(d)(\text{I})(0))$

The initial value of a dimension is 0.

PD: PARAMETERDECL → **Store** → **Store**		
PD⟦ AP; PD ⟧σ	=	**PD**⟦ PD ⟧(**P**⟦ AP ⟧σ)
PD⟦ ϵ ⟧σ	=	$Id(\sigma)$

P: PARAMETER → **Store** → **Store**	
P⟦ I : P [L,U] H ⟧σ	=
let $(o, si, se, d) = \sigma$ **in** $(o, si,\ new(se)(\text{P})(\text{H},\text{I},(I(\text{L})),\text{U},\text{false},\epsilon),d)$	
P⟦ I : P [L,U] det U H ⟧σ	=
let $(o, si, se, d) = \sigma$ **in** $(o, si,\ new(se)(\text{P})(\text{H},\text{I},(I(\text{L}),\text{U},\text{true},\epsilon),d)$	
P⟦ I : P H ⟧σ	=
let $(o, si, se, d) = \sigma$ **in** $(o,\ new(si)(\text{P})(\text{H},\ \text{I},\ \epsilon),\ se,\ d)$	

The two definitions for sequence parameters differ only in the value of the determines field, which is true for determining parameters and false otherwise.

Table 3.6: Abstract syntax of arguments

ARGUMENTS			ARGUMENT			STRINGS		
AS	::=	-A AS	A	::=	O	NS	::=	S NS
	|	ϵ		|	O S		|	ϵ
				|	O NS			

where

O $\in$ **Ide**,
S $\in$ **String**,
NS $\in$ **StrSeq**

3.4.7 Semantics of the constraints

We give the abstract syntax of the constraints in table 3.5. The valuation function **CS** is defined as follows:

CS : CONSTRAINTS $\rightarrow$ **Store** $\rightarrow$ **Boolean**$_\perp$		
CS$[\![$ C ; CS $]\!]\sigma$	=	**if C**$[\![$ C $]\!]\sigma$ **then CS**$[\![$ CS $]\!]\sigma$ **else** *false*
CS$[\![$ ϵ $]\!]\sigma$	=	*true*

CS just checks all constraints in order, until one proves to be false.

3.4.8 Semantics of the instantiation arguments

Arguments for a **TEMPL** expression modify the store. As we will see, it is not easy to describe this modification in a non-clumsy way.

AS indicates that arguments are processed one by one, and that the order is important in the sense that previous arguments can be overwritten by later ones.

AS : ARGUMENTS $\rightarrow$ **Store** $\rightarrow$ **Store**		
AS$[\![$ ϵ $]\!]\sigma$	=	$Id(\sigma)$
AS$[\![$ -A AS $]\!]\sigma$	=	**AS**$[\![$ AS $]\!]$(**A**$[\![$ A $]\!]\sigma$)

Table 3.7: Abstract syntax of TEMPL

TEMPL
TEMPL ::= DS CS [] S

A : ARGUMENT → **Store** → **Store**
$\mathbf{A}[\![\ \text{O}\ ]\!]\sigma = \textbf{let}\ \sigma = (o, si, se, d)\ \textbf{in}$
$(select(o)\text{O}, si, se, d)$
$\mathbf{A}[\![\ \text{O S}\ ]\!]\sigma = \textbf{let}\ \sigma = (o, si, se, d)\ \textbf{in}$
$((select(o)\text{O}),$
$(\lambda i.\ \textbf{if}\ option(si)(i) = \text{O}$
$\textbf{then}\ ((help(si)(i), o, s)$
$\textbf{else}\ si(i)),$
$(\lambda i.\ \textbf{if}\ option(se)(i) = \text{O}$
$\textbf{then let}\ (h, o, l, u, b, v) = se(i)\ \textbf{in}\ (h, o, l, u, b, cons(s)(\epsilon))$
$\textbf{else}\ se(i)),$
$d)$
$\mathbf{A}[\![\ \text{O NS}\ ]\!]\sigma = \textbf{let}\ \sigma = (o, si, se, d)\ \textbf{in}$
$((select(o)\text{O}),$
$(\lambda i.\ \textbf{if}\ option(si)(i) = \text{O}$
$\textbf{then}\ (help(si)(i), o, hd(\mathbf{NS}[\![\ \text{NS}\ ]\!]))$
$\textbf{else}\ si(i)),$
$(\lambda i.\ \textbf{if}\ option(se)(i) = \text{O}$
$\textbf{then let}\ (h, o, l, u, b, v) = se(i)\ \textbf{in}\ (h, o, l, u, b, \mathbf{NS}[\![\ \text{NS}\ ]\!])$
$\textbf{else}\ se(i)),$
$d)$

NS : STRINGS → **StrSeq**
$\mathbf{NS}[\![\ \epsilon\ ]\!] = \epsilon$
$\mathbf{NS}[\![\ \text{S NS}\ ]\!] = cons(\text{S})(\mathbf{NS}[\![\ \text{NS}\ ]\!])$

As the reader can see, **A** is not such an easy function to read. However, the principles are simple. Each different production rule contains an option. This option is selected in every store that is created. In the store that we have constructed, it is not possible to determine directly which option declares what parameter. It is also not possible to determine whether the values provided after the option are meant for a singular or for a sequence parameter.

Our solution to this problem is, that we modify both the store for singular and for sequence parameters at the position where the option O is the declaring option of the parameter. In the case that a sequence parameter is given only one value we construct a sequence of strings containing only that value and in the case that a singular parameter is given a list of values we take the head of that list to be the wanted value.

3.4.9 Semantics of instantiation

The abstract syntax for a TEMPL expression is given in table 3.7.

Before we can instantiate anything, we must provide default values for those parameters that have not been given a value by the arguments.

defaults : **Store** → **Store**

$defaults = \lambda s.\textbf{let}\ (o, si, se, d) = s\ \textbf{in}$
$(o,$

$(\lambda i.$**let** $(h, o, v) = si(i)$ **in**
if $v = \epsilon$
then $(h, o, Str(i))$
else $(h, o, v))$,
$(\lambda i.$**let** $(h, o, l, u, b, v) = se(i)$**in**
if b
then (h, o, l, u, b, v)
else $(h, o, l, u, b, append(v)(ln(v))(d(u))(Str(i)),$
$d)$

In this function we see the use of the *append* function on sequence parameters; it appends default values which are made unique by attaching a (index) number to the string representation of the identifier of the parameter.

Note that both the options and the dimensions in the store are unmodified by this function. Dimensions are set using the function *det*:

det : **Sequence** → **Ide** → **Dimension** → **Dimension**
$det = \lambda s.\lambda i.\lambda d.$**let** $(h, o, l, u, b, v) = s(i)$ **in**
if b
then $[u \mapsto (ln(v) - l + 1)]d$
else $[u \mapsto d(u)]d$

It is defined rather tricky: The store of sequence parameters is applied to the identifiers in this store to obtain the value of those identifiers. If the determines boolean b is set, then we modify the upperbound dimension u in d to the length of the values of i, minus its lowerbound plus one. Otherwise, the value of the dimension remains unchanged.

Now we have enough functionality to describe instantation:

Inst : TEMPL × ARGUMENTS → $\mathbf{String}_\perp$
Inst⟦ (DS CS [] S, AS) ⟧ =
let σ = **AS**⟦ AS ⟧(**DS**⟦ DS ⟧ *empty*) **in**
if **CS**⟦ CS ⟧σ
then let $(o, si, se, d) = \sigma$ **in** **S**⟦ S ⟧(*defaults*$(o, si, se, det(se)$(**dom**$(se))(d)))$
else $\perp$

We see that instantiation takes a pair (TEMPL, arguments) and returns a string. The store argument that S needs is created by valuating the declarations with an empty store, applying the argument valuation to this store, determining the dimensions, after which the defaults can be added to this store.

3.5 Pragmatics

Let us recall that the pragmatics of a language describe the aspects of ease-of-use of the language, its effectiveness in achieving its stated goals and the effort to implement the parser/evaluator of its expressions.

Ease-of-use has been a major design requirement in this system as it should not distract attention from the prime use of the tool: to aid in the specification in COLD of large designs. We will explain why we chose the specific representation of TEMPL keywords and symbols.

The issue of effectiveness allows us to discuss our choices of the constructs available in the language from several alternatives.

For the effort needed to implement the language syntax and semantics, we restrict ourselves to the use of TEMPL in conjunction with COLD as the source language. We discuss how our language design decisions are reflected in the implementation issues.

3.5.1 Language symbols

In general, a language manipulates symbols. Let us call these symbols the *data symbols* of the language, which consists in the case of TEMPL of option symbols, parameter symbols, dimension symbols, local iteration symbols, integer constant symbols and string constant symbols.

In order to manipulate these data symbols, the language must provide operations which are also described using symbols. Let us call these symbols the *operator symbols*. Examples of the *representations* of these operator symbols in TEMPL are [, |, +,) and {.

The editor and keyboard technology that has been more or less standardized in the last few decades, restricts our choice of representations for the symbols to sequences of ASCII characters.

For example, in COLD all strings of printable characters can be a representation of some data or operator symbol, see e.g. [3] , appendix A, or a combination of these symbols. This implies that whatever representation we use for TEMPL data and operator symbols, they can conflict with the representation of COLD data and operator symbols.

Other representations

These conflicts could be resolved if we could use e.g. different typefaces for TEMPL symbols and COLD symbols. We could for example write the following expression in $\texttt{TUP}_n$:

`[tup?  FUNC tup : [i,1,n|Item[i]| # ] -> Tup]`
as:
[tup? FUNC tup : [i,1,n|Item[i]| #] -> Tup]
or even, by using different typefaces for different classes of TEMPL symbols, as
[tup→ **FUNC tup** : $\Sigma_{i=1}^{n}$(**Item**$_i$ ∘ #) -> **Tup**]

From this example we see that although option tup and parameter tup share the same representation as a sequence of characters, they can be distinguished on their resp. typefaces. One could even imagine the use of colours to distinguish symbols.

The last representation is written in LaTeX as follows:

```
[{\it tup}$\to$
\verb|    FUNC |{\bf tup}\verb| : |$\Sigma^{n}_{i=1}$(
{\bf Item$_i$}$\circ$\verb| # |)\verb| -> |{\bf Tup}]
```

A parser would have no problem in parsing the above. If the writer has to create such strings however, some kind of graphical editor would come in handy.

Choosing operator symbols

The reason for the above digression may not be immediately clear. The point that we want to make is that if one defines the symbols of a language from scratch, as for example with COLD, then any keyboard symbol can be used and given an unique meaning. Then, using a graphical notation could be clumsy and redundant for the experienced specifier. However, in the case of TEMPL, which is in essence a macro language, it could provide a better overview on the source language.

The macros are defined on the COLD data and operator symbols. We have chosen to use one construct { *literal string* } by which the TEMPL writer can include any string that will not be evaluated by the evaluator in the TEMPL expression. The symbols { and } have been chosen

because they are the comment brackets of COLD. As TEMPL is meant to replace the frequently used library components, which contents is supposed to be known, comments in the TEMPL expressions will probably not occur frequently, especially because comments have no meaning to the analysis tools. Another reason is that they are clearly brackets, as opposed to for example ''. The literal string symbols { and } can themselves be escaped by prefixing them with a \. So, the comment {comment...} in COLD can be generated by \{{comment...}\} in TEMPL. The other comment facility, %, remains available unmodified, so all comments can still be expressed in TEMPL expressions.

Fortunately, we have chosen our other operator symbols from a small set of characters, and have constructed them so that little COLD text will have to be bracketed.

The opening character for the TEMPL constructs is [. The corresponding closing character is]. These brackets are used in COLD to indicate parameterisation of COLD identifiers. As such, they occur not too commonly in arbitrary COLD texts. Another reason for the choice of this symbol is that it cannot be used as a user-defined operator symbol in the COLD-texts, as opposed to e.g. +, @.

As a separation symbol between the body and the separator expressions of the iteration expression and the then- and else-expressions of the if-then-else- expression we have chosen the character |. The advantage of this character is that it visually separates two expressions as a kind of hedge. A disadvantage of this symbol is its use as a possible separator of expressions in COLD, representing choice. The other character that has the look of a separator is #. However, as we have seen in the example for TUP_n, it occurs frequently as the separation expression itself.

Choosing data symbols

We will discuss the data symbols mentioned in the beginning of this paragraph in separate paragraphs.

Identifier symbols The identifier symbols used in TEMPL have a representation described by the following regular expression:

$$\{a, \ldots, z, A, \ldots, Z\}^+\{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, \_\}^*$$

This representation is the usual one for identifiers in any common language and poses no unusual restrictions on the TEMPL writer. However, identifier symbols should not conflict with the TEMPL operator symbols. So, the use of AND as the representation of an identifier symbol is not allowed.

Option symbols Option symbols are elements of the set of identifier symbols. Recognition of identifier symbols as option symbols is easy, as they occur only in two constructs of the TEMPL language: the option declaration and in the constraint of the choice expression.

Parameter symbols Parameter symbols are elements of the set of identifier symbols. Parameter symbols should not conflict with TEMPL operator symbols. It is advised not to use COLD keyword symbols either, because the representation of a parameter symbol can be used as its default value. Because of this default mechanism, it is advised to use as representation, the representation that one would choose when one writes a COLD specification.

Parameter symbols, option symbols and dimension symbols may have the same representation. This is possible, because as above explained, the context in which the representations are used determines their symbol class.

The possible conflicts between parameter representations and string constant representations are discussed in the paragraph on string constants.

Dimension symbols Dimension symbols are elements of the set of identifier symbols.

Local iteration symbols The iteration symbols are elements of the set of identifier symbols. However, they have an associated scope with which they can be distinguished. Each iteration expression opens a scope in which a local iteration symbol is declared. After evaluation of the iteration expression, the scope is closed.

Integer constant symbols The integer constant symbols of TEMPL only occur in contexts where their representations are easily recognized as such.

String constant symbols String constants are of course very important in TEMPL. They define the pattern in which a substitution takes place. Basically, one could say that the entire actual TEMPL expression is a sequence of string constants, interleaved with TEMPL constructs.

This is however a too simple view on the TEMPL expression, as string constants can occur within the TEMPL constructs and spaces and newlines do have effect on the workings of the instantiation.

String constant symbols have a representation described by the following regular expression:

$$\{!, \ldots, Z, \hat{}, \ldots, z, \tilde{}\}^+$$

Note that this regular expression overlaps the one that described identifiers. To distinguish the two, one can choose one class of symbols and somehow adapt their representation such that the overlap disappears. For example, one could use ' around the string constants and the overlap disappears. In our example the result would be the following:

```
[tup?'    FUNC 'tup' : '[i,1,n|Item[i]|' # ']' -> 'Tup]
```

Conversely, one could put a & character, which is not a representation of a COLD-keyword, before parameter symbols.

```
[tup?    FUNC &tup : [i,1,n|&Item[i]| # ] -> &Tup]
```

However, as one can see, both these approaches have the disadvantage of reducing the readability of the TEMPL expressions[3]. Furthermore, they introduce the use of new TEMPL operator symbols.

The approach that we have taken differs rather dramatically from the above approaches. Globally, we propose the following, a more technical presentation of our approach can be found in subsection 3.5.3.

First of all, all symbols are separated by spaces and newlines. The spaces and newlines are significant for the lay-out of the instantiation, so they are read as a format expression. Then, if a symbol falls in the class of parameter symbols, which is determined by the declaration of the parameters, then this symbol is a parameter. Otherwise, the symbol is a string constant.

Formatting symbols Whereas a normal programming language would ignore spaces and newlines, TEMPL does not. At least, not in the actual expression part of the TEMPL expression. Formatting symbols have a representation described by the following regular expression:

$$\{\mathtt{NL} + \mathtt{SP}\}^+$$

Note that tabs (HT) are ignored. This is done because the meaning of a tab is not standardized. Sometimes it is used as an indentation referring to some word on the previous line, at other times it means a fill of space to the next tab stop, where tab stops have been defined as fixed (but arbitrary) positions in a line of text. Vertical tabs (VT)[4] and form feeds (FF) are also ignored as formatting symbols.

[3] It is still possible if wanted to use such a distinction by means of the literal text construct. This would lead to the following:

```
[tup?{ FUNC }tup{ : }[i,1,n|Item[i]|{ # }]{ -> }Tup]
```

[4] If the reader has some use for the VT as a formatting symbol, please let me know.

3.5.2 Effectiveness of TEMPL

Now we have discussed our representations of our data and operator symbols, we want to show how the language achieves its effectiveness as a macro-extension to COLD.

The options

The options are given at the beginning of the TEMPL expression. These options can be used to indicate certain choices for the instantiation of the TEMPL expression, for example an option `fold` might indicate that the sort that is exported from the component is not to be defined in the current instantiation, but in an imported component of this instantiation, using a renaming.

We say that all options are *free*, by which we mean that there are no standard options in the language. For example, we could have made the option `fold` a build-in option, because of its possible frequent use. We have chosen not to do so, in order to give the TEMPL developer more freedom of expression and to keep the instantiation process as transparent as possible (no surprises).

Initially, all options are deselected. A default setting of options would require in the instantiation language not only an expression for option selection, but also an expression for option deselection. Furthermore, the specifier should not only know the possible options, but also which of these options are default options. The necessary selection of an option is now indicated in the constraints section of a TEMPL expression.

Constraints on options An arbitrary selection of options can cause a nonsense instantiation. To ensure a right collection of selected options is used for the instantiation, the TEMPL developer can indicate logical constraints on options. The constraints state whether a certain collection of selected options is valid for the instantiation or not. For example, in $\texttt{TUP}_n$ we state that the options `fold` and `unfold` may not be selected at the same time (`fold <=> (NOT unfold)`). By this constraint, we actually mean that we do not allow an instantiation in which a sort is imported and a sort is defined with the same name.

Constraints could also have been expressed in a if-then-construct, with the constraints as the condition and the whole actual TEMPL expression as its then-expression. In this way, unsatisfied constraints would lead to a textually empty instantiation. However, by explicitly stating the constraints in the declaration part of a TEMPL expression, unsatisfied constraints can be treated in a different fashion, so the system can indicate more clearly why the instantiation failed.

Substituting values

In a TEMPL expression, it is possible to declare and use string parameters. These parameters can be given a string value, or *argument*. If a parameter is not given a value, then its default value is substituted when the parameter is referenced.

We have chosen to allow two kinds of string parameters: *singular* parameters, which can be given only one value and *sequence* parameters, which can be given a finite sequence of values. Singular parameters do not really introduce new possibilities for instantiation. Renaming a specific name is already possible with the current way of instantiation and parameterisation. A singular parameter symbol has as its default value its string representation.

The new sequence parameters allow more genericity to the parameterised components. Instead of writing 25 generic components for `TUP1,...,TUP25`, one for each different number of parameters, TEMPL allows one to write just one TEMPL expression for $\texttt{TUP}_n$ in which the n is determined by the number of values given for a certain sequence parameter. That is what is really meant by saying that "p determines n". The default value of a sequence parameter is its string representation, extended with the string value of its index.

Note that this index may not always begin with 1, see for example $\texttt{ENUM}_n$ in appendix E, in which a lowerbound of 0 is used for a sequence parameter. This has of course consequences for the determination of the dimension, so in general, the dimension has the value of the number of values provided minus 1 plus the lowerbound of the determining parameter.

The need for a dimension n is given by the iteration construct, which is closely related to sequence parameters.

Iteration constructs The iteration construct allows a number of similar string patterns to be generated separated by some other (possibly empty) expression. This number is usually related to a dimension, but this is not required. In this pattern, almost always a reference is made to a sequence parameter by means of an index.

One could imagine a very implicit iteration construct, such as

```
[Item[1] # ... # Item[n]]
```

It has the advantage of being an intuitive notation for

```
[i,1,n|Item[i]| # ]
```

We have chosen the last construct for the following reasons:

- We have noted that a reference to the current iteration identifier may be useful in the body of the construct apart from being an index.
- The nesting of iteration constructs becomes more explicit.
- The last construct is much easier to recognize and parse. In the first example above, the separation symbol # is recognized as it occurs twice in the construct, one time directly before the ... and one time directly after ... It is certainly not trivial to let a parser recognize this.

We have adapted our representation of the iteration from the one used in VLib, an infinite library of LOTOS specifications, described in [13]. Our iteration construct can be simpler, because COLD is based on conditional equations, as opposed to the equational style used in LOTOS. The iteration construct is a very powerful construct, as any sequence of expressions can be used as its body and as its separation. However, the local iteration identifier may not be referenced in the separation, because then the question arises what the value of the local identifier might be. Is it the same value as in the body, 1/2 added to this value or one added to this value? In any case, we will never know.

Default values Our approach has been that the parameters symbol representation is also its default value. Explicitly indicating a default value may be useful when, for example, distinct parameters symbols have a different representation but may have the same default value. It is common in COLD to overload simple identifiers, because names are distinguished on these identifiers *and* their functionality. However, if a string parameter is not given a value, then its functionality is usually not exported. Its real name is thus hidden for external reference, so neither interesting, nor important.

A specifier can however still indicate the same values for different parameters, thus effectively restoring the situation.

A default name which contains e.g. operator symbols can not be represented now, because the parameter names are simple identifiers. However, an examination of the typical parameterisation used in the IGLOO library reveils that this is not a real manco.

Providing arguments for an instantiation

How easy is it to provide values for an instantiation? There are two ways of providing arguments.

Easy interaction For easy interaction, a clicking interface is used. From the declaration part of a TEMPL expression, enough information can be derived to build a GUI in which the specifier can indicate his arguments.

In the help text of an option or parameter, information relating to the use and purpose of this option or parameter can be provided. The specifier can view this information, thus aiding him effectively in providing arguments.

Fast interaction For fast interaction, it is possible to provide arguments on the command-line. This is especially interesting if the specifier is familiar with the TEMPL that he wants to instantiate or if the instantiation process is to be automated.

The option names can be used as a way of providing command-line flags. Stating an option on the command-line indicates the selection of an option. If an option is also used to declare a parameter then value(s) for this parameter can be provided after the option name. This trivial syntax allows the recognition of instantiation arguments to be quite a straightforward matter.

Mapping arguments to parameters

As we have seen in the semantics section, it is not trivial to map the instantiation arguments to parameter values. We have to take determining parameters, default values and constraints on options into account.

If the dimensions were to be known beforehand, we could easily generate default values for the parameters, and overwrite these with the arguments if present. However, we have chosen to let the dimensions to be implicitly determined by the number of values given for a determining parameter.

This implies that we first have to calculate the dimensions from the number of values given for the determining sequence parameters, after which we can add default values to the other sequence parameters if necessary.

3.5.3 From concrete to abstract syntax

Before we can evaluate a TEMPL expression, we must have some knowledge about the structure of its contents. For example, if we encounter the opening symbol [of a TEMPL construct, we must know which] closes the construct. So, we have to split up a TEMPL expression into a sequence of the expressions mentioned in the abstract syntax.

We have used the compiler-generator tool Elegant ([8, 1]) for this purpose. Although Elegant clearly has interesting facilities, we encounter problems that are not easily solvable in Elegant. We will mention the most important ones in the following paragraphs.

Scanning The scanning phase of language analysis is concerned with the grouping of characters in the expression to symbols for the parser. This is achieved by describing symbols by regular expressions of characters. In this way, each sequence of characters can be mapped to one of these classes. If all these classes are disjoint, then a scanner is called *deterministic*. In general, this is not the case. For example, identifier and keyword classes usually overlap. In the description of a scanner, one can usually give a preference to the recognition of one symbol class over the other.

Spaces and newlines We have chosen to let spaces and newlines to be significant in the actual TEMPL expression. However, they are not important in the declaration part of the TEMPL expression, except for the separation of symbols. The scanner itself is translated from a highly abstract description to C-code which is fixed during the scanning of a TEMPL expression. We would like to change the character sets that form the classes of symbols that the scanner recognizes, just before the actual expression is scanned.

Unfortunately, this is not possible in the high-level description language that is used to describe the scanner. The solution that we have chosen for this problem is that we always scan spaces and newlines, but ignore them in the declaration and control parts of the TEMPL expression.

Declaration keywords A more or less converse problem lies in the symbols that we use as keywords and symbols in the declaration part of a TEMPL expression, such as `AND`, `DIMENSIONS`, `;` etc. These symbols should not be read as keywords in the actual TEMPL expression. Just like the above problem, it cannot be solved in the scanner definition.

Our solution to this problem is to explicitely include extra production rules in the parser that recognize these keyword symbols as literal expressions.

Several types of strings In TEMPL expressions, we can distinguish several types of strings. We have the *literal* string symbol, the *COLD* string symbol, the *identifier* string symbol and the *format* string symbol.

A literal string is easily recognized because of the matching `{` and `}`. The characters that form the class of identifier symbols are however contained in the characters that form the class of COLD string symbols, which, in its turn, is contained in the literal string symbol class. So, they overlap rather dramatically.

We have chosen to give identifier symbols precedence over COLD string symbols. This means, for example, that `add(one, zero)` is split into seven separate symbols. An identifier symbol `add`, then a literal symbol (see above discussion) `(` followed by another identifier symbol `one`, then a literal symbol `,` then a format symbol followed by another identifier symbol `zero` followed by a literal symbol `)`. This is sensible, because each of the identifiers `add`, `zero` and `one` could be a parameter. In a following paragraph, we will explain how the parser combines this symbol information to create the abstract syntax expressions.

Parsing The parsing phase of language analysis uses the symbol information created by the scanner to determine the production rules that were used to build the expression. A useful characteristic of production rules is that they are *LL(1)*, which means that an efficient parser can be built that needs only one symbol to determine which production rule was used.

Recognizing substitution We recognize an identifier symbol as a singular substitution if the identifier is declared as a parameter. Otherwise, we transform the identifier to a piece of literal text.

In the production rule that recognizes sequences of expressions, we combine the head and the tail of an expression sequence to a sequence substitution only if the head expression is a singular substitution, the following tail expression is a value expression and these expressions are not separated by spaces or newlines.

Overloaded keywords Because the symbol `[` is used to recognize many constructs, there are a number of production rules that are needed to find out which construct is actually used. After the opening square bracket, we can expect some clues by which we can determine what construct follows. Say we see an identifier. If it is followed by a `]`, then we know that this was a value construct. If it is followed by a `,,` then we know that this is an iteration construct. If it is followed by a `?`, then we know it opens a choice construct. If it is followed by a `[`, we know that we have recognized a function symbol.

If we don't see an identifier, but e.g. a `(`, then analogues processing takes place to determine whether the construct is a value-expression or a choice expression, or more precisely, whether a MATHEXPR or a CONSTRAINT should be recognized. The `(` indicates that a compound expression using TEMPL data and operator symbols follows.

Special purpose language analysis The specific structure of a TEMPL expression is best and most efficiently exploited with a special-purpose language analyzer, which combines the scanning and parsing phase of language analysis, using the knowledge that is difficult to express in Elegant. However, construction of such a language processor would take considerably more time than the one currently written in Elegant.

We will give some issues which this processor could take into account:

- Different uses of keywords and symbols in the declaration and expression part.
- Pattern matching on strings to determine whether they are parameters or strings.
- Reading the text as a whole into memory, where abstract syntax string expressions are represented as pointers to positions in this text.

3.6 TEMPL summary

To conclude this chapter on the language TEMPL, we will give here a brief summary of the main characteristics of TEMPL.

A TEMPL expression can be viewed as a function with parameters. In this view instantiation is function application on the instantiation arguments. This is the natural way of looking at a macro language.

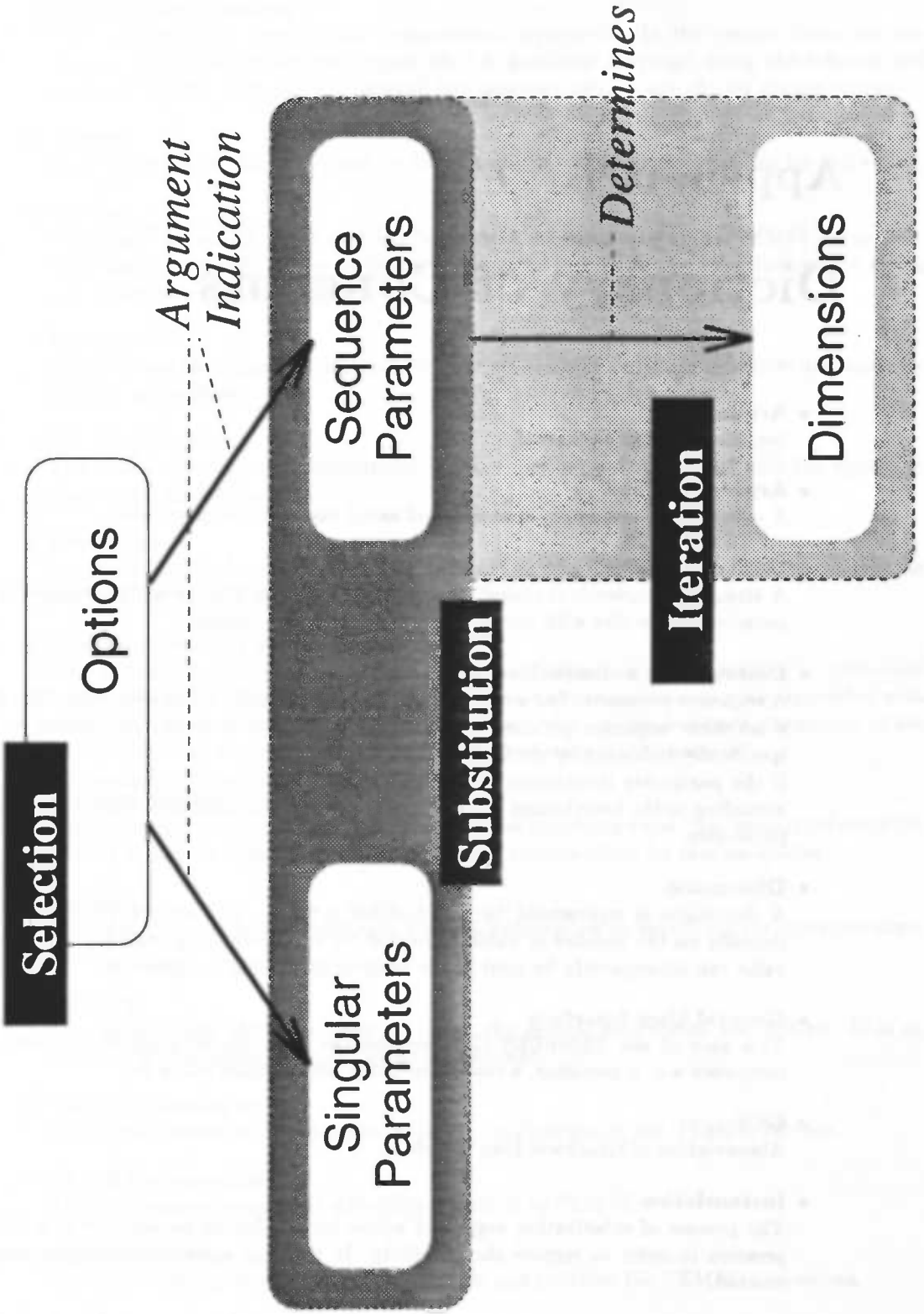
In the expression itself there is no assignment, which means that parameters have only one value during the evaluation of the TEMPL expression. The default value mechanism for string parameters ensures that these parameters always have a value.

There are only a few language constructs which are orthogonal to a large extend, which implies that there is little cognitive overload for the writer of TEMPL expressions. By overloading TEMPL symbols we have aimed at keeping an overview of the source language.

We have aimed at giving control over the instantiation to both the TEMPL writer, the person that writes TEMPL expressions and the instantiator, the person that instantiates the TEMPL expressions. At one hand the instantiator wants to be able to use arbitrary selection and substitution. At the other hand the TEMPL writer can restrict the actions of the instantiator by means of constraints and the dependencies of the sequence parameters via the dimensions.

Our choice of language constructs (selection, iteration and substitution) are tightly bound to the parameters that make are used in these constructs. Furthermore, as indicated in figure 3.1, the parameters themselves are also linked to one another.

So, in short, we have created a small but powerful language of which constructs and parameters and expressiveness and restrictions are balanced.



Legend : The typeface of a word indicates its meaning:
TEMPL language construct
TEMPL parameter
Dependencies between parameters

Figure 3.1: Overview of TEMPL parameters and their dependencies

Appendix A

Dictionary of Concepts

- **Arguments**
  See *instantiation arguments.*

- **Arguments Library**
  A collection of previously specified and saved instantiation arguments.

- **Declaring option**
  A string parameter is declared by an option. In this way, providing values for a string parameter coincides with the selection of its declaring option.

- **Determining a dimension**
  A sequence parameter has a dimension as its upperbound. It can determine this dimension if no other sequence parameters share this dimension as their upperbound or if this is specifically indicated by the `DETERMINES` keyword.
  If the parameter determines a dimension, then the value of this dimension is calculated according to the lowerbound for this parameter and the number of values provided for this parameter.

- **Dimension**
  A dimension is represented by an identifier symbol. The integer value of a dimension depends on the number of values provided for its determining sequence parameter. This value can subsequently be used in the body of the TEMPL expression.

- **General User Interface**
  This part of the TEMPLES GUI remains the same for all possible argument sets. It comprises a.o. a menubar, a results area, the file selection boxes etc.

- **GUI**
  Abbreviation of Graphical User Interface.

- **Instantiation**
  The process of substituting argument values for the formal parameters of a TEMPL expression in order to remove the genericity. In this way a non-generic component can be created.

- **Instantiation arguments**
  A set of strings used to indicate the values of the TEMPL expression parameters for an instantiation.

- **Instantiation Language**
  The language that specifies the *instantiation arguments*. In the system there are two distinct languages to provide arguments: A graphical language using check-boxes and textfields via the GUI and a command-line language using a specific yet simple syntax.

- **Library**
  A library is a collection of objects, in which a specific object can be searched for and edited.

- **Option**
  An option indicates a choice that can be made in the instantiation of a TEMPL expression. This choice is related to the inclusion of pieces of text or to the substitution of a string parameter (see *declaring option*).

- **Parameters**
  TEMPL uses three kinds of parameters: *options*, *singular (string) parameters* and *sequence (string) parameters*.

- **Private Library**
  This is the collection of instantiations that the specifier has contructed with the system to provide for his own needs.

- **Resources**
  A description of the attributes of the parameters of a TEMPL expression to be used for the building of a TEMPL specific user interface.

- **Sequence (string) parameter**
  This is a parameter that can obtain a sequence of values, each of which can be referenced by an index. The identifier symbol associated with the sequence parameter appended with a variable index is at the same time its default value, if this sequence parameter is not substituted with argument values. See also *determining a dimension*.

- **Singular (string) parameter**
  This is a string parameter that can obtain at most one string value. The associated identifier symbol is also its default value if no argument value is given for this parameter.

- **Specifier**
  This is the person who instantiates TEMPL expressions by specifying instantiation arguments.

- **TEMPL**
  This is the name of the language in which the generic expressions are written. It is an acronym of The Easy Macro-Processing Language.

- **TEMPL developer**
  This is the person who writes new TEMPL components for the TEMPL Library.

- **TEMPL expression**
  This is a generic component definition described in TEMPL.

- **TEMPL Library**
  This is the library that is used in the system and contains the TEMPL expressions.

- **TEMPL Structure**
  This is the result of parsing a well-formed TEMPL expression. It represents the information needed for the instantiation process.

- **TEMPL specific User Interface**
  This is the part of the TEMPLES GUI with a specific lay-out for the parameters of the current TEMPL expression. See also *resources*.

- **TEMPL User Interface Generator**
  This is a part of the system that can construct the TEMPL specific user interface from the resources of a TEMPL expression.

- **User Interface**
  The concept of a user interface is usually related to a human-centered view on information processing. This means that the user controls the application, rather than letting the application control the user.

Appendix B

Description of used Tools

For the development of the system we have used software development tools, of which we give here an introductory description.

B.1 Elegant

Elegant is the name of both a high-level programming language and a compiler generator system. An introduction to Elegant is given in [8] and a more thorough description in [1].

The programming language has features taken from object-oriented, imperative and functional programming languages.

The most evident object-oriented feature is that Elegant allows subtyping and polymorphism.

The functional programming style is present in the availability of pattern matching, lazy evaluation and the ZF-expressions.

The imperative aspect of Elegant is present in the use of global and local variable declarations, the assignments on these variables and the sequential execution of statements with side-effects.

The compiler generator Elegant allows the compiler writer to write a highly abstract BNF-like description of a grammar to be automatically converted into a scanner description and a parser description. These are frameworks, to which user defined actions can be added.

The programming language Elegant is used to describe these actions. To allow for modular design of the compiler, extended makefile technology is used. If one adapts a specific development directory lay-out where the different files are stored, then the compiling of the entire compiler is reduced to the simple command `make`.

B.1.1 The programming language

We will give below an example of our code with which we can explain certain useful aspects of Elegant:

```
SPEC UNIT Parameter
```

Elegant uses two kinds of files: `SPEC` files and `IMPL` files. In the `SPEC` file, the exported types and functionality are listed, which is translated to C header files. The exported types and functionality can be imported into other units.

```
TYPE

ABSTRACT
  Parameter      = ROOT, prefix : Ident
                       , option : Ident
                       , help : String
```

We introduce the *abstract type* Parameter. Each string parameter of TEMPL has an identifier, called prefix here, a declaring option, option and a help string, help. An Elegant object may not have an abstract type, so we introduce the following *subtypes*:

```
SequenceParameter < Parameter, lwb : Int
                             , upb : Ident
                             , values : List (String)
                             , determines : Ident

SingularParameter < Parameter, value : String
```

Both types SequenceParameter and SingularParameter are subtypes of Parameter. They have the fields given in Parameter in common, but differ in their private fields.

The following constant is used to indicate undefinedness:

```
CONSTANT

  Undefined : String = ""
```

We specify the following function by giving its functionality:

```
RULES

  wf:[pmrs : List (Parameter)] : Bool
```

The implementation unit describes how the exported functionality is defined. In the implementation unit we can also use functions etc. that are private to the unit.

```
IMPL UNIT Parameter

RULES

{--------- Defined:[pmr : Parameter] : Bool ---------}
(*
   <Defined> returns true if <pmr> is defined
   on all its values ;
   <Defined> returns false if <pmr> is not defined
   on some of its values
*)

  Defined:[pmr : SingularParameter] : Bool
  -> RETURN (pmr.value # Undefined)
  ;

  Defined:[pmr : SequenceParameter] : Bool
  LOCAL l : List (String)
          = { v | v : String <- pmr.values, ( v == Undefined ) }
        b : Bool = (l = NIL)
        c : Bool
          = (DimValue:[pmr.upb] - pmr.lwb <= (#pmr.values))
  -> RETURN (b AND c)
  ;
```

With the function Defined, we see *polymorphism* in action. It is a private function, which checks whether a Parameter is defined on all its values. This check differs for objects with type SequenceParameter and objects with type SingularParameter.

The code for a SingularParameter is simple. We just check if the value field is not Undefined (# denotes unequality).

For a SequenceParameter we check whether all its values are defined, by using a ZF-expression on the list of strings, values. The list l constructed by this expression, contains those strings that are undefined. If this list is NIL, we can assume that all values are defined. Next, we check whether the number of values in the values list is greater than or equal to the number of values actually needed.

```
{--------- wf:[pmrs : List (Parameter)] : Bool-------------}
(*
   <wf> returns true if all elements of pmrs are defined,
   otherwise it returns false
*)

  wf:[NIL : List (Parameter)] : Bool
  -> RETURN (TRUE)
  ;

  wf:[pmrs : List (Parameter)] : Bool
  -> IF Defined:[pmrs.head]
     THEN RETURN (wf:[pmrs.tail])
     ELSE RETURN (FALSE)
     FI
  ;
```

We can simply make a list of some type by surrounding the type with List (...). A list has as its fields a head and a tail. The field head has as its type the base type and the field tail has as its type List (base type). In this way we can simply compute the value of wf.

Note that the way we have defined the functions is probably not the most efficient or elegant definition. However, the code is completely clear to the experienced Elegant reader.

B.1.2 The compiler generator

We take a piece of our grammar and see how this is translated into the final attribute grammar and scanner.

The following text is a part of the temples.bnf file, which is the basic file from which the parser and scanner are generated. It uses an extended form of BNF-rules

```
    .
    .
    .
DimensionDeclaration ::= "DIMENSIONS" Dimensions .

Dimensions ::= Dimension // ";" .

Dimension  ::= ident .
```

Elegant can transform this into a rudimentary attribute grammar and scanner definition, which follows:

```
ATTRIBUTE GRAMMAR temples

GRAMMAR

ROOT
   .
   .
   .
  DimensionDeclaration ()
  -> DIMENSIONSsym ()
     Dimensions ()
  CHECKS
  LOCAL
  ;

  ALTDimensions () -> ;

  ALTDimensions ()
  -> semicolonsym ()
     Dimensions ()
  CHECKS
  LOCAL
  ;

  Dimensions ()
  -> Dimension ()
     ALTDimensions ()
  CHECKS
  LOCAL
  ;

  Dimension ()
  -> identsym ()
  CHECKS
  LOCAL
  ;
```

In the CHECKS section of the above rules checks can be made on the attributes, for example whether the dimension was redeclared. In the LOCAL section semantic actions can be specified, for example to declare a dimension. In the () one can specify the IN and OUT attributes of the grammar rules.

The compiler writer may himself complete this framework.

```
ATTRIBUTE GRAMMAR temples

  Dimension.Declare    AS DimDeclare
  Dimension.IsDeclared AS DimIsDeclared

GRAMMAR

ROOT

  DimensionDeclaration ()
```

```
  -> DIMENSIONSsym ()
     Dimensions ()
  CHECKS
  LOCAL
  ;

  ALTDimensions () -> ;

  ALTDimensions ()
  -> semicolonsym ()
     F ()
     Dimensions ()
  CHECKS
  LOCAL
  ;

  Dimensions ()
  -> Dimension (OUT id : Ident)
     ALTDimensions ()
  CHECKS
     IF DimIsDeclared:[id]
     THEN "Dimension " + String:[id] + " already declared"
  LOCAL
     b : Bool = DimDeclare:[id, 0]
  ;

  Dimension (OUT id)
  -> identsym (OUT id : Ident)
  CHECKS
  LOCAL
  ;
```

We see that little action has to be performed. We must of course import the used functionality and rename them for clarity. The `identsym` rule has the `OUT` attribute `id : Ident`. This rule is specified in the scanner which follows later.

This `id` is transported upwards to the rule `Dimensions`. There, we check whether this `id` is already in use as a dimension identifier. If so, we write a diagnostic message to standard error.

`DimDeclare` returns a boolean, so we assign it to `b`. As this is an example, we will not go into details on how the dimension is declared, nor will we discuss the reason why this code may give problems.

The generated scanner looks like this:

```
SCANNER temples

RULES
  .
  .
  .
  DIMENSIONSsym   ::= "DIMENSIONS" .

  semicolonsym    ::= ";" .

  identsym = .
```

```
SETS
```

The keyword DIMENSIONSsym is defined with the literal given in the temples.bnf file, the same can be said about the semicolonsym. The identsym is still undefined. With the use of SETS we can define what an identsym is:

```
SCANNER temples

RULES
  .
  .
  .
  DIMENSIONSsym   ::= "DIMENSIONS" .

  semicolonsym    ::= ";" .

  identsym : Ident   =  Letter { Letter | Digit | "_" }  .

SETS

  Letter      ['a'..'z', 'A'..'Z']
  Digit       ['0'..'9']
```

The symbol Ident indicates that an OUT attribute should be generated with type Ident. Actually, this is a function call to the function Ident which turns a sequence of characters into an identifier.

After SETS we can indicate classes which are assigned to the scanned characters.

The resulting parser

All components are translated into C-files. This implies amongst others that it is relatively easy to include specific C-code in the resulting parser. The other main advantage is that portability to other platforms is in principle simple.

The generated C-files are then compiled into executable object-code. A disadvantage of Elegant is in our view the time it takes to build the compiler.

B.2 The OSF/Motif GUI-toolkit

For the GUI we have chosen to use the OSF/Motif-toolkit, which provides a large number of customizable elements which can be used in the interface, such as buttons, editable text windows, directory browsers, menu bars etc. Two main approaches exist to the development of Motif GUI's:

- Using a GUI generator tool like Xdesigner
- Using C to specify the lay-out of the interface

In the next two subsections we will discuss these possibilities.

B.2.1 X designer

This tool allows fast development of Graphical User Interfaces and is based on the Motif toolkit. It is easy maintainable and a lot of trivial work in writing a toolkit-based user interface is automated, so it reduces the development time of a GUI. The application is controlled by including

callbacks, which specify the actions that the application can perform. These callbacks must be written by the developer himself. So, it clearly separates application from interface.

However, it does require the need to learn the way in which it can be used. It also lacks the flexibility that is required for the interface. Our application as it is wanted here doesn't really need for complicated layouts, instead, it uses repititions of similar components. Besides that, and this is the most important argument, the number of interface-elements is different for each TEMPL-specific GUI, which is not easy to model in XDesigner.

Although XDesigner is not used in the implementation, it does allow fast, interactive construction of an interface, so we have used it to develop and test the 'look and feel' of different interfaces.

B.2.2 Motif and C

The Motif toolkit consists of a comprehensive library of utility routines. In general, a *widget*, the name of an interface element, has a window, attributes associated with that window, may contain other widgets and a number of actions or *events* that the widget can handle. The library is set up in an object-oriented fashion, in the sense that widgets are defined in a hierarchy and widgets *inherit* attributes from their ancestors.

The library system is set up in such a way that most attributes have a default value. This value is either compile-time known or run-time known, the last being the case if the widget *inherits* some values from its parent widget.

The default values may be changed by the interface designer so he can build his own specific interface. The possible values and their typing can be found in e.g. [7], or in the UNIX manual pages. Furthermore, the actions can be specified to perform some wanted action of the application.

We will give an example of our code here without comment, and refer to [2] or [7] for an introduction to toolkit programming.

```
void DoSaveInstantiation ()
/*** Display a dialog box by which the user can
     indicate whether he wants to save the last
     instantiation or not.

************************************************/

#include "xtem.h"

/* top_level ************************************
   ---------
This widget is used as the father of all the main
dialogs in this file.
                                                */
extern Widget top_level ;

/* DoSaveInstantiationYesCB *********************
   -----------------------
This callback routine unmanages its corresponding
widget and calls SelectFile with tag
SAVE_INSTANTIATION so the user can select a file
in which he can save the results of the last
instantiation.
                                                */
```

```
void DoSaveInstantiationYesCB (w, unused, call_data)
Widget w ;
caddr_t *unused ;
XmAnyCallbackStruct *call_data ;
{
  XtUnmanageChild (w) ;

  SelectFile ( SAVE_INSTANTIATION ) ;

  return ;
} ;

/* DoSaveInstantiationNoCB ************************
   ----------------------
This callback routine just unmanages its
corresponding widget and returns.
                                                  */
void DoSaveInstantiationNoCB (w, unused, call_data)
Widget w ;
caddr_t *unused ;
XmAnyCallbackStruct *call_data ;
{
  XtUnmanageChild (w) ;

  return ;
} ;

/* DoSaveInstantiation ****************************
   -------------------
This procedure creates and manages a simple dialog
in which the user is asked whether he wants to save
the last instantiation.
                                                  */
void DoSaveInstantiation ()
{
  Arg      al[10] ;
  register int ac ;
  Widget   msg ;
  XmString xmstring ;
  XmString xmyes ;
  XmString xmno ;

  xmyes = XmStringCreateLtoR ("Yes",
                    (XmStringCharSet)XmFONTLIST_DEFAULT_TAG) ;
  xmno = XmStringCreateLtoR ("No",
                    (XmStringCharSet)XmFONTLIST_DEFAULT_TAG) ;
  xmstring = XmStringCreateLtoR ("Save last instantiation?",
                    (XmStringCharSet)XmFONTLIST_DEFAULT_TAG) ;

  ac = 0 ;
  XtSetArg (al[ac], XmNmessageString, xmstring) ; ac++ ;
  XtSetArg (al[ac], XmNokLabelString, xmyes) ; ac++ ;
```

```
  XtSetArg (al[ac], XmNcancelLabelString, xmno) ; ac++ ;
  msg = XmCreateTemplateDialog (top_level, "information", al, ac);
  XmStringFree (xmstring) ;
  XmStringFree (xmyes) ;
  XmStringFree (xmno) ;

  XtAddCallback (msg, XmNokCallback,
                 (XtCallbackProc) DoSaveInstantiationYesCB,
                 (XtPointer) NULL) ;

  XtAddCallback (msg, XmNcancelCallback,
                 (XtCallbackProc) DoSaveInstantiationNoCB,
                 (XtPointer) NULL) ;

  XtManageChild (msg) ;

  return ;
};
```

Appendix C

User's manual of TEMPLES

The TEMPLES system is based on the following two executable programs:

- `temples` : This program executes the instantiation process on its provided TEMPL expression argument with the provided instantiation arguments.
- `xtemples` : This program is a generic, easy-to-use instantiation arguments editor, enveloped around `temples`, with capabilities to easily inspect the results of the instantiation process.

In this chapter we describe how these programs interact with eachother and their usage. We recommend that the reader who is new to the world of instantiation should read subsection C.3.2 first.

C.1 `temples`

`temples` has the following synopsis:

```
temples [ -res ] templ-file-name instantiation-arguments
```

The option `-res` indicates that a file named `Tresources` should be generated in the working directory. This file is of no direct interest for the user, except that it allows the system to generate a specific user interface for the instantiation arguments available for `templ-file-name`

If the TEMPL file name is given without a directory indication, then it is assumed that the file is present in the working directory.

If the syntax of the TEMPL expression in the templ-file is not correct, then an indication of the line and column where the syntax was violated and a short description of this syntax error will be written on the standard error.

C.1.1 Files

The following ASCII-files can be generated in the working directory by `temples` as a result a call to `temples`:

- `Tevaluation`
  This file contains the evaluation of the TEMPL expression with the given arguments. This file is generated only if no syntax errors occured, the constraints of the TEMPL expression were satisfied, the flag `-res` was not set and no static semantic errors occured, by which we mean references to unknown identifiers, invalid index ranges etc.

- `Tresources`
  This file contains information that is used by `xtemples` to generate the generic contents of its window. This file is only generated if the `-res` flag is set. This file is not generated if syntax errors occured in the declaration part of the TEMPL expression.
- `Terrors`
  This file contains the static semantic errors that occured during the parsing and evaluation of the TEMPL expression. For a description of the possible error messages, see subsection C.2.4.
- `Twarnings`
  This file contains messages that may be errors but that did not make evaluation of the TEMPL expression impossible. For a description of the possible warning messages, see subsection C.2.4.
- `Timports`
  This file lists the component names that are imported in the instantiation. Currently, this has not been implemented. You are refered to the `IMPORT` section of the instantiated component for this information.
- `Tconstraint_errors`
  This file lists the constraints that were not satisfied by the provided option arguments.

Note that all these files have been prefixed with a `T`, which makes it easy to discard them if needed.

C.1.2 Arguments

The instantiation arguments that you can give for the TEMPL expression are given as follows:
{*-option-name* { *string-values* } }
How many values and options you can give depends on the TEMPL expression that you instantiate in the following way:

- If an option-name declares no parameter, then no string-values should be given. The option-name will be selected.
- If an option-name declares a singular parameter, then one string-value should be given, separated by space. The option-name will be selected and the string-value will be the value of the declared parameter.
- If an option-name declares a sequence parameter, then at least one string-values should be given, separated by space. The string-values will be given to the sequence parameter in the following order: The first string-value will be given to the sequence parameter at its lowerbound index, the next string-value to its lowerbound + 1 index etc.

  Note that one or more string-values *must* be given if the sequence parameter determines its dimension.

As you can see, following this way of instantiation does require you to know the declaration part of the TEMPL expression to some detail.

C.2 xtemples

`xtemples` is the Motif graphical user interface for the TEMPLES system. It has the following synopsis:

`xtemples`

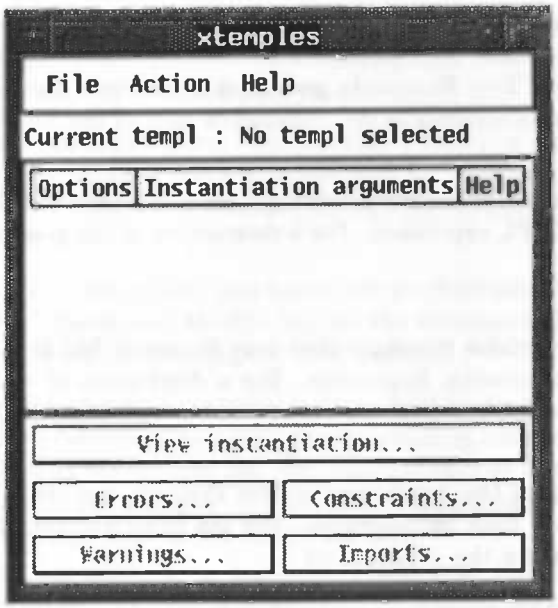


Figure C.1: Startup screen of `xtemples`

C.2.1 The interface

The window shown in figure C.1 will appear on your screen. At the top of the window is the menubar. Each of the submenu options (*File*, *Action* and *Help*) can be selected with the mouse to make a pulldown menu appear.

Below the menubar is a line that indicates the current TEMPL you are working on. On startup, this line will indicate that you have not yet selected a TEMPL.

The generic arguments dialog is given in a scrolled window. On startup, this window will be empty.

Listed below the generic arguments dialog are a number of pushbuttons, which allow you to see the results of the instantiation. Each of these five buttons will become sensitive to mouse selection if the corresponding result was created.

The *File* menu

The file menu option allows you to choose from a number of actions.

- The *Select TEMPL* menu option
  This option pops up a file selection box in which you can choose a TEMPL file for which you want to provide arguments.
- The *Select .args* menu option
  This option pops up a file selection box in which you can select an already created arguments file.
- The *View any file* menu option
  This option pops up a file selection box in which you can select any ASCII file name of which the contents will be displayed in a dialog.
- The *Save .args* menu option
  This option pops up a file selection box in which you can select or enter a file name to which your arguments must be saved.

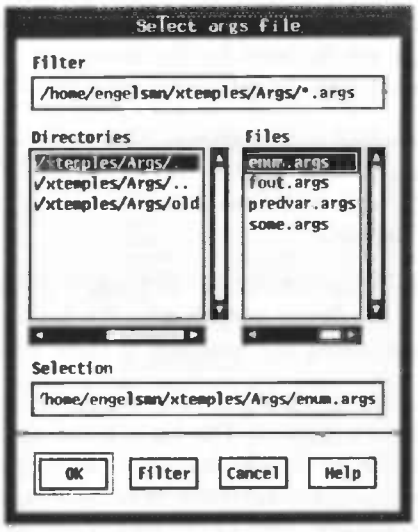


Figure C.2: File selection in `xtemples`

- The *Save last instantiation* menu option
  This option pops up a file selection box in which you can select or enter a file name to which the last instantiation result must be saved.

- The *Exit xtemples* menu option
  This option ends the `xtemples` session.

File selection

An example of the file selection box is given in figure C.2. The *Filter* window indicates which file names will be listed. In the *Directories* window you can select the directory in which you want to search for files. By clicking the *Filter* button, this directory change takes effect.

You can select a file by clicking on its name in the *Files* window. You can also type in the full path of the file in the *Selection* text field.

The button *Cancel* lets you return without action. Pressing the button *Ok* will let `temples` analyze the file indicated in the *Selection* textfield, in order to create a specific arguments dialog.

The generic arguments dialog

Let us say that you have selected the TEMPL file `enum.templ`. This will modify the generic dialog window. Usually, scrollbars will appear on the lower and right side of this window. You can either scroll through the arguments dialog or adjust the size of the `xtemples` window until the scrollbars disappear.

After adjusting the size, you will see something like figure C.4. The current templ line indicates that you are now working on the TEMPL *enum*.

We identify three main columns in the arguments dialog. They are headed by the labels *Options*, *Instantiation arguments* and *Help*.

Options The column listed below Options consists of a number of so-called *check-boxes*. You can select a check box by clicking on the label or by clicking on the square. These labels are the *option names* from the TEMPL file. An option is selected if the checkbox is black, otherwise it is deselected. We see that, initially, all options are deselected.

Instantiation arguments In the column headed by *Instantiation arguments* you can indicate values for the parameters that are declared by the preceding option. You can see that there are two (or, more precisely, three) ways of indicating parameter values.

- A *text field* is listed after the option. This indicates that you can provide one value, but note that this value is only used if you have selected the preceding option. You can edit the textfield by selecting it with the mouse and typing in the wanted value. It is not necessary to finish with a Return.
- A *text field* is listed after the option, and below this textfield are a large *scrolled list* window and the *buttons Add*, *Delete* and *Modify*. We list the functionality of the textfield and the scrolled list by examining the workings of these buttons:
  - ⋆ The button *Add* appends the value in the text field to the list at the position just after the highlighted posistion. The highlighted position is indicated by the black bar with the white letters in the scrolled list. The functionality of *Add* can also be achieved by entering Return in the textfield. If there is no highlighted position, then *Add* appends the value in the text field to the end of the list.
  - ⋆ The button *Delete* removes the highlighted item from the list and the highlight disappears. If no item in the list is highlighted, then pressing *Delete* has no effect.
  - ⋆ The button *Modify* removes the highlighted item and places it in the textfield, where it can be edited. The modified item is inserted at its previous position in the list by pressing *Add*. If no item is highlighted, then pressing *Modify* selects the last item in the list to be modified.

  Any item can be selected by the mouse. If there are many values in the list, then scrollbars will appear that allow you to see the entire list.
- No way to indicate values at all. This is the case when an option doesn't need values for a parameter.

Help In the column headed by *Help*, there are listed a number of *Help* buttons. Each *Help* button will, when pressed, display a dialog as in for example figure C.3. In this dialog a message appears which may aid you in determining the function of the parameter of option in the instantiation of the TEMPL file. You can remove the dialog by pressing the *Ok* button.

The *Action* menu

In figure C.6 a filled arguments dialog are shown for the TEMPL enum. `xtemples` allows only one action on these arguments: by selecting *Instantiation* from the *Action* pulldown menu you can instantiate the TEMPL file enum with these arguments. We have executed the instantiate action with the arguments specified in figure C.6. The instantiation results are indicated by the pushbuttons below the arguments dialog. You can see that the pushbutton *Constraints...* has

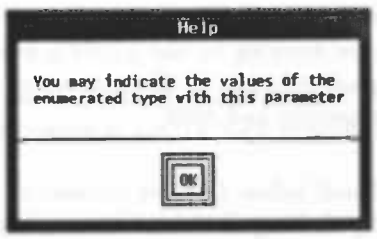


Figure C.3: A help dialog

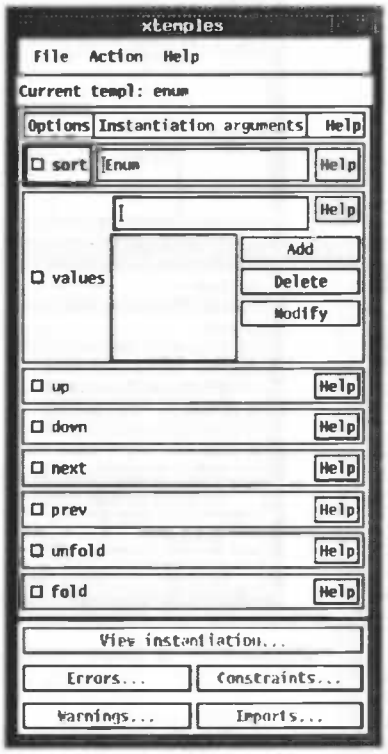


Figure C.4: An empty arguments dialog

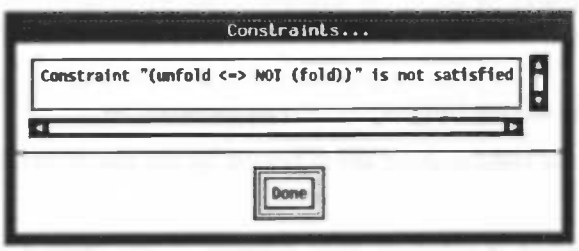


Figure C.5: A constraint has not been satisfied

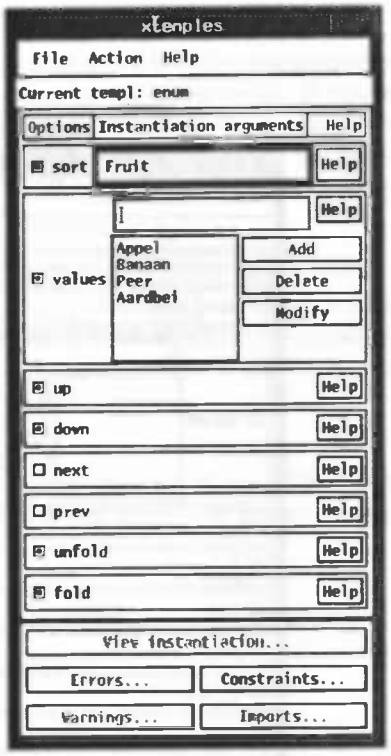


Figure C.6: A filled arguments dialog

been made sensitive, while the other buttons are not. Pressing *Constraints...* causes the dialog shown in figure C.5 to appear.

Apparently our set of selected options was not allowed by the constraints. We can use the information provided by this dialog to try again, with the option "fold" deselected this time. The results of the instantiate action are this time indicated by the *View Instantiation...* button. Selecting this button will cause the dialog listed in figure C.7 to appear, which displays the instantiated component.

If an instantiation was created by `temples`, then a dialog appears in which you are prompted whether to save this instantiation or not. The last instantiation was stored in a temporary file which is removed each time a new instantiation takes place. If you select the *Yes* button of this dialog a file selection box appears in which you can indicate the filename to which the instantiation should be saved. If you select the *No* button, no action is undertaken. You can always save the alst instantiation by selecting the *Save last instantiation* menu option.

Results of the instantiation

The results of the instantiation are indicated by the sensitivity of the buttons listed below:

- *Errors...*
  If this button is sensitive, then pressing it will cause a window with error messages to appear. They are generally related to the TEMPL expression that you want to instantiate.

- *Warnings...*
  If this button is sensitive, then pressing it will cause a window with warning messages to appear. The system is not sure whether these messages are errors or whether you intended these problems. They were at least not severe enough to make the instantiation impossible.

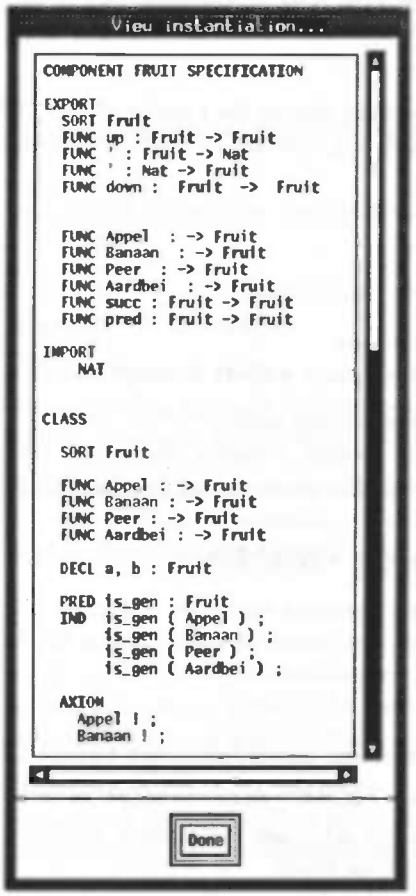


Figure C.7: The instantiated component

- *Imports...*
  If this button is sensitive, then pressing it will cause a window to appear which contains the components that were imported by this component. If you are going to use this component in a design, then you should make sure that these components are present. In the current implementation, this button will never be sensitive, because this feature has (not yet) been implemented. If you want to see the imported components, you should check the `IMPORT` section of the instantiated component.

- *Constraints...*
  If this button is sensitive, then pressing it will cause a window to appear which contains messages describing the unsatisfied constraints. Constraints are expressed on options, and you should modify your selection of options in such a way that the unsatisfied constraints will become satisfied, i.e. true.

- Syntax Errors
  If syntax errors occured during the parsing of the TEMPL expression, then no instantiation takes place and a dialog appears in which the syntax errors are listed. There is no button for syntax errors.

The *Help* menu

The *Help* menu provides two choices:

- The *About* menu option
  This menu option displays a short message about `xtemples`

- The *Help on xtemples* menu option
  This menu option creates a dialog containing roughly the same information as in this section, but without the pictures and in plain ASCII.

C.2.2 Environment variables

`xtemples` uses some environment variables. Here we list them with an indication of their use, an example of how they can be set in your `.login`-file and what their value is if the variable is not set:

- `TEMPLES`
  The directory where the parser/evaluator `temples` can be found.

  Example : `setenv TEMPLES /usr/local/bin/temples`

  If this variable is not set, then the current working directory will be used as the directory where `temples` can be found.

- `TEMPLES_ARGS`
  The startup directory where the argument files can be found.

  Example : `setenv TEMPLES_ARGS $HOME/temples/Args`

  If this variable is not set, then the current working directory will be used as the startup directory for the argument files. The search pattern for argument file names is `*.args`

- `TEMPLES_TEMPL`
  The startup directory where the TEMPL expression files can be found.

  Example : `setenv TEMPLES_TEMPL $HOME/temples/Templ`

  If this variable is not set, then the current working directory will be used as the startup directory for the TEMPL expression files. The search pattern for TEMPL file names is `*.templ`

- `TEMPLES_RESULTS`
  The startup directory where the instantiation files can be found.

  Example : `setenv TEMPLES_RESULTS $HOME/temples/Inst`

  If this variable is not set, then the current working directory will be used as the startup directory for the argument files.

- `PWD`
  The path of the working directory at startup.

  This variable is updated by the shell, so there is no need to set it in your `.login` file.

  The working directory is used to hold the temporary files of the system and is the default directory for the above mentioned directories if their resp. environment variable is not set.

C.2.3 Files

In addition to the files mentioned with **temples**, **xtemples** uses the following temporary files:

- `Tsh.tmp`
  This file contains a shell-script that is used for the communication between **temples** and **xtemples**.

- `Targs.tmp`
  This file contains an instantiation call to **temples**.

- `Tsyntax_errors`
  This file contains the syntax errors that occured during the parsing of the TEMPL expression if its size is greater than 0, otherwise no syntax errors occured.[1]

The file `Thelp` contains general help information that is displayed by by selecting the *Help on xtemples* menu option from the submenu *Help*. It contains an ASCII version of the text in this section and should be present in the same directory as **temples**.

C.2.4 Diagnostics

Error messages can be generated as a result of the instantiation. They can be found in the file `Terrors`, or viewed by selecting the **errors...** button of **xtemples**. The error messages are usually related to the TEMPL expression, not to the instantiation itself:

- **Unknown value for dimension** *d*
  This error occurs if you try to use a dimension that has not been given a value. This can occur when you do not provide values for the determining parameter of *d* or if you have declared *d*, but you did not let it be determined by a sequence parameter.

- **Reference to undefined singular parameter** *p*
  This error occurs if you try to insert the value of *p* in the instantiation, but its value is not defined.

- **Reference to undefined sequence parameter** *p*
  This error occurs if you try to index a parameter *p* that has not been declared as a sequence parameter.

- **Attempt to address undefined value**
  Somehow the index you provided for a sequence parameter was out of its defined range of values. The index was either too high or too low.

[1] The reason why the size of the file is relevant is a consequence of the use of this file as a standard error redirection of the **temples**-call.

- **p is not well-defined**
  This error indicates that a singular parameter p is undefined or a sequence parameter p has some of its needed values undefined.

- **Option o already declared**
  You have declared more than one option with the same identifier.

- **Dimension d already declared**
  You have declared more than one dimension with the same identifier.

- **Parameter p already declared**
  You have declared more than one parameter with the same identifier.

- **Option o not declared**
  You have made a reference to option o, but you have not declared it as an option.

- **Dimension d not declared**
  You have made a reference to dimension d, but you have not declared it as a dimension.

- **Identifier i not in scope or not declared**
  You have referenced i in some value expression, but you did not declare it either as the local identifier or as a dimension. Another possibility is that you referenced i in the separator part of an iteration construct, which is not allowed.

The following warning messages can be generated as a result of the instantiation. They can be found in the file `Twarnings`, or viewed by selecting the *warnings...* button of `xtemples`. These warning messages are usually related to the instantiation of a TEMPL expression:

- **Option o declares a parameter, but no values were given for this parameter**
  This warning occurs if you have selected an option that is used to declare a parameter and thus acts as a flag for its values. However, these values were absent. In the instantiation the default value will be used.

- **Only one value provided for sequence parameter p**
  This warning indicates that only one value was found for the sequence parameter p. Usually, this is not an error, but for certainty we have included it as a warning.

- **Option o does not declare a parameter, argument discarded**
  You have listed a value after option o, but option o was not used in the TEMPL expression as a declaring option. So, this value is discarded. Note that the option remains selected.

- **More than one argument provided for p; assuming first value as argument**
  This warning will be given if you have provided more than one value for the singular parameter p. The first value of this list is used as the value for p.

The following warning messages are related to the TEMPL expression itself:

- **Upperbound of p unequal to determined dimension; assuming d as the determined dimension**
  This warning is generated if you have declared a sequence parameter of which the upperbound dimension is unequal to the determined dimension. `temples` assumes that the upperbound d is the determined dimension.

- **Redefinition of i as iteration index**
  This warning is generated if you have used as a local identifier in an iteration construct an identifier i which is already in scope.

- **Unknown function *f*, ignoring function call**
  You have indicated a call to an unknown function *f*.

- ***i* too few arguments for function *f*, ignoring function call**
  This warning occurs if you have specified too few arguments for the function *f*.

The syntax errors are usually self-explanatory, the given row and column coordinates of the error aid in the quick removal of the error.

The following error message leads to immediate exit from the tool:

- **Error opening file *f*, exiting**
  This error indicates that somehow a file could not be opened for reading. It should not occur frequently, and could be the result of wrong file modification rights on *f*.

C.3 Notes

In the development of the tool, we have distinguished two kinds of users of the system: The TEMPL developers, the users that write TEMPL expressions and the specifiers of instantiators, the users that instantiate TEMPL expressions for their private use.

We want to give some specific information for both these users here.

C.3.1 Notes for the TEMPL developer

A TEMPL developer should make sure that his TEMPL expressions can indeed be correctly instantiated. To this end we recommend that you should first use the `temples` evaluator without instantiation arguments to check the syntax of your TEMPL expressions. If the syntax is correct, then the `xtemples` interface can be used to provide arguments for your TEMPL expression and to see how these influence the instantiation.

Note that the declaration part of a TEMPL expression is only changed if it is re-read. The instantiate action has no effect on the contents of the arguments dialog. For example, if you have read in a TEMPL expression and the arguments dialog is displayed, then modifying the help-text of a parameter will take effect in the dialog until you re-read the TEMPL expression.

Note that the help-text of a declaring option is superseded by the help-text of the parameter that it declares.

We recommend that you read the TEMPL manual thoroughly, because some language constructs operate quite subtle, for example function calls.

A final remark is that you should provide sufficient help information in order to allow the instantiator to procede with his work in a fast and easy manner. He should be able to instantiate a TEMPL expression with the aid of these help texts alone.

C.3.2 Notes for the instantiator

We will give here an introduction to the world of instantiation.

Instantiating a *generic* text allows you to customize this generic text to your own, private needs. In the TEMPLES system we use *TEMPL expression* as our terminology for a generic text.

The system operates in a similar way to a macro-preprocessor and is to some extend comparable with the C preprocessor. In a TEMPL expression, three kinds of parameters can be defined:

- *Option parameters*, used to select or deselect pieces of text in the TEMPL expression. Another use for options is that they *enable* string parameter substitution, an aspect discussed in subsection C.1.2.

- *String parameters*, which exist in two kinds:
  - ⋆ *Sequence string parameters*, which are used in the TEMPL expression to allow a number of similar substitutions to take place.
  - ⋆ *Singular string parameters*, which are used in the TEMPL expression to allow exactly one substitution to take place.

All these parameters can be given a value or *instantiation arguments*. These instantiation arguments are fixed during the *evaluation* of the TEMPL expression. Parameters have a default value, which means that options are by default not selected and the string parameters have by default some unique string value.

Because the arbitrary use of options can give rise to non-sense instantiations, the TEMPL developer can express *constraints* on these options which indicate which set of selected options can be used for the instantiation. If the selected options do not satisfy these constraints, then instantiation is not possible.

You can provide the arguments with the interface of `xtemples`. This manual explains how this is done.

The system is set up in such a way that you do not need to examine the parameter list of the TEMPL expression yourself, nor need you inspect the remainder of the TEMPL expression to determine the function of the parameters. Usually, the help-texts given with the parameters should provide you with enough information to instantiate meaningfully.

We hope that you will achieve the level of experience and knowledge that allows you to instantiate components with `temples` directly, without the use of the interface. How this is done is explained in subsection C.1.2.

This concludes our summary description of instantiation.

Appendix D

User's manual of TEMPL

The user's manual of TEMPL contains the sections 3.2, 3.3 and 3.4 of this thesis with some minor adaptions.

Appendix E

TEMPL examples

E.1 TEMPL example : Enum

In this example we see how the arithmetic expressions in the indexing constructs can be made to work for us. Furthermore, we see that in the first axiom of the class section, distinctness of the enumerated values is indicated with the aid of two nested iteration expressions.

```
OPTIONS
  up      {} ;
  down    {} ;
  next    {} ;
  prev    {} ;
  values  {} ;
  sort    {} ;
  unfold  {} ;
  fold    {}

CONSTRAINTS
  values ;
  sort   ;
  unfold <=> NOT (fold)

DIMENSIONS
  n

PARAMETERS
  sort : Enum      {} ;
  values : x[0,n]  DETERMINES n {}

[]
COMPONENT Enum SPECIFICATION

EXPORT
  [sort          ?SORT Enum]
  [up            ?FUNC up : Enum -> Enum]
  FUNC ' : Enum -> Nat
  FUNC ' : Nat -> Enum
  [down          ?FUNC down :  Enum  ->  Enum]
```

```
  [prev          ?FUNC prev :  Enum  ->  Enum]
  [next          ?FUNC next :  Enum  ->  Enum]
  [values?[i,0,n|FUNC x[i]   : -> Enum |
  ]]
  FUNC succ : Enum -> Enum
  FUNC pred : Enum -> Enum

IMPORT
    NAT
  [fold?, ENUM[n+1] RENAMING [sort?{Enum}[n+1] TO Enum]
              [values?, [i,0,n|{x}[i] TO x[i]|
              , ]]
         END
  ]

CLASS

  [unfold?SORT Enum

  [i,0,n|FUNC x[i] : -> Enum |
  ]

  DECL a, b : Enum

  PRED is_gen : Enum
  IND  [i,0,n|is_gen ( x[i] ) |;
       ]

  AXIOM
    [i,0,n|x[i] ! |;
    ] ;
    [i,0,n|[j,i+1,n|x[i] /= x[j] |;
    ]|;
    ]is_gen (a)

  FUNC ' : Enum -> Nat
  IND  [i,0,n|'( x[i] ) = [i] |;
       ]

  FUNC ' : Nat -> Enum
  IND  [i,0,n|'([i]) = x[i] |;
       ]

  FUNC succ : Enum -> Enum
  IND  succ ('(a)) = '(b) => succ (a) = b

  FUNC pred : Enum -> Enum
  IND  pred ('(a)) = '(b) => pred (a) = b

  THEOREM
    '(a)! ;
    succ(a)! <=> a /= x[n] ;
```

```
    pred(a)! <=> a /= x[0]

  ]
  [up?FUNC up : Enum -> Enum
  IN   i
  OUT  j
  POST i =  x[n] => j = x[n]
     ; i /= x[n] => j = '(i + 1)
  ]

  [down?FUNC down : Enum -> Enum
  IN   i
  OUT  j
  POST i =  x[0] => j = x[0]
     ; i /= x[0] => j = '(i - 1)
  ]

  [next?FUNC next : Enum -> Enum
  IN   i
  OUT  j
  POST i =  x[n] => j = x[0]
     ; i /= x[n] => j = '(i + 1)
  ]

  [prev?FUNC prev : Enum -> Enum
  IN   i
  OUT  j
  POST i =  x[0] => j = x[n]
     ; i /= x[0] => j = '(i - 1)
  ]

END
```

E.2 TEMPL example : Stack

In the following example we see how easy a component can be extended with TEMPL parameters without too much effort. Note that the CLASS section of the component is even the same as in the IGLOO definition.

```
OPTIONS
  Stack       {} ;
  Item        {} ;
  new         {} ;
  push        {} ;
  pop         {} ;
  top         {} ;
  max_size    {} ;
  cur_size    {} ;
  is_empty    {} ;
  is_full     {}
```

```
CONSTRAINTS
  Item

PARAMETERS
  Stack : Stack
  Item  : Item
  new   : new
  push  : push
  pop   : pop
  top   : top
  max_size : max_size
  cur_size : cur_size
  is_empty : is_empty
  is_full  : is_full

[]
COMPONENT [uc[Stack]] SPECIFICATION

EXPORT
  SORT Stack
  [new      ?PROC new : Nat -> Stack]
  [push     ?PROC push : Stack # Item ->]
  [pop      ?PROC pop : Stack ->]
  [top      ?FUNC top : Stack -> Item]
  [max_size?FUNC max_size : Stack -> Nat]
  [cur_size?FUNC cur_size : Stack -> Nat]
  [is_empty?PRED is_empty : Stack]
  [is_full ?PRED is_full : Stack]

IMPORT
  NAT,       % interface
  SEQ\[Item\] % representation

CLASS

  SORT Stack              VAR
  FUNC seq : Stack -> Seq VAR

  DECL s : Stack
     , i : Item

  FUNC max_size : Stack -> Nat

  PRED SI :
  DEF FORALL s
      ( seq (s)!
      ; len(seq(s)) <= max_size (s)
      )

  AXIOM SI

  AXIOM INIT => NOT EXISTS s ()
```

```
PRED is_empty : Stack
IND  seq (s) = empty => is_empty (s)

PRED is_full : Stack
IND  len(seq(s)) = max_size(s) => is_full (s)

FUNC cur_size : Stack -> Nat
IND  cur_size(s) = len(seq(s))

FUNC top : Stack -> Item
IND  hd(seq(s)) = i => top(s) = i

PROC new : Nat -> Stack
IN   n
OUT  s
PRE  TRUE
SAT  NEW Stack
POST seq(s) = empty
   ; max_size (s) = n

PROC push : Stack # Item ->
IN   s,i
PRE  len(seq(s)) < max_size (s)
SAT  MOD seq(s)
POST seq(s) = cons(i, seq'(s))

PROC pop : Stack ->
IN   s
PRE  LET b := tl(seq(s))
SAT  MOD seq(s)
POST seq(s) = b

% definedness

THEOREM top(s)! <=> NOT is_empty(s)
      ; cur_size (s)!
      ; max_size (s)!

END
```

Appendix F

Testing the system

Testing a program doesn't show its correctness, but it can indicate its incorrectness. We should keep this paradigm from E.W. Dijkstra in mind when discussing the testing of the system. We can say that we have found the system not to be incorrect by means of our tests. In this chapter we will discuss how we tested the system with a focus on the testing of the parser/evaluator `temples`. We have first tested `temples` and `xtemples` separately and then tested the system as a whole.

F.1 Testing `xtemples`

We have made some resource files `Tresources` that `xtemples` uses to build its TEMPL specific arguments dialog. Our main concern of this testing was that the file was read in correctly. This was checked by verifying that all options, parameters and help texts were indeed correctly displayed at their wanted positions. Next, we tested whether all default values indicated in the `Tresources` file were also mapped to the corresponding interface elements. The user interface elements themselves were thoroughly tested on their correct behaviour. The writing of arguments to a file was also tested, by comparing the contents of this file with the arguments in the user-interface.

F.2 Testing `temples`

Testing temples consists of three main parts. First of all, the *syntax* should be parsed correctly. This is in prinicpal trivial to test. Second, we have to make sure that the actual parameters are mapped correctly to the formal parameters. This includes the determining of dimension values, the default values if no actual parameters were given and the correct parsing of the instantiation arguments. Thirdly, *evaluation* of the parsed expression should be done correctly.

The problem with the testing of a language lies in the fact that language constructs may *depend* on eachother. For example, it is not possible to test whether the values for a sequence parameter are filled in correctly without being sure that the enabling option is selected.

A global overview of our test set is given in the following list:

1. Checking the formatting and plain text.
   A normal text of printable characters is given in the TEMPL expression. There is no declaration part. Either none or some escape sequences are used to include special TEMPL-characters in the text.

2. Selection of options.
   A list of options is given, and in the actual TEMPL expression each of these options is the

condition in an if-then-expression with as its true branch the name of that option.

3. Singular parameter setting and defaults.
   A list of singular parameters is declared and their corresponding options. The parameters are listed in the actual TEMPL expression. The result should be a list of parameters, of which the selected ones have their new value and the unselected ones have their default value.

4. Overloading COLD identifiers and TEMPL identifiers.

5. Sequence parameter setting and defaults.
   In this test one sequence parameters is declared. In the TEMPL expression all its values are listed using the iteration construct.

6. Sequence parameter setting and defaults.
   In this test two sequence parameters is declared. Their lowerbound and upperbound are identical and one determines the dimension. In the actual TEMPL expression all their values are listed using the sequence construct.

7. Sequence parameter setting and defaults.
   In this test two sequence parameters are declared. Their lowerbound is identical but their upperbounds differ. Both determine their dimension. In the actual TEMPL expression all their values are listed using the sequence construct.

8. Arithmetic expressions.
   With the use of two sequence parameters, which both receive arbitrary arguments, the sequence construct and the arithmetic expressions are tested.

9. Nested choice constructs.
   A nested number of similar choiceconstructs is given in the actual TEMPL expression. Their corresponding options are declared.

Each of these tests consists of one (sometimes more) TEMPL expression and one or more sets of arguments. The tests do not necessarily have to result in an instantiation, as we consider the warnings, errors and constrainterrors equally significant as possible results of the tests.

Let us examine in more detail for example test 6. In test 6, the following TEMPL file is used:

```
OPTIONS
  een   { };
  twee  { }

DIMENSIONS
  n

PARAMETERS
  een  : enen[1,n] DETERMINES n { };
  twee : tweeen[1,n]            { }

[]
[i,1,n|enen[i]| ]
[i,1,n|tweeen[i]| ]
```

The mentioned purpose of test 6 can be divided in the following subtests, in this case only the provided arguments are relevant:

1. No arguments are provided, both options are not selected.

2. No arguments are provided, only the option of the determining parameter is selected.
3. No arguments are provided, only the option of the non-determining parameter is selected.
4. One argument is provided, for the determining parameter. Its option is selected
5. Some arguments are provided, for the determining parameter. Its option is selected
6. One argument is provided, for the non-determining parameter. Its option is selected
7. Some arguments are provided, for the non-determining parameter. Its option is selected
8. Both options are selected. The determining parameter receives one argument, the other none.
9. Both options are selected. The non-determining parameter receives one argument, the other none.
10. Both options are selected. Both parameters receive one argument
11. Both options are selected. Both parameters receive an arbitrary, equal number of arguments
    (a) Both options are selected. Both parameters receive two arguments.
12. Both options are selected. The determining parameter receives more arguments than the non-determining parameter
13. Both options are selected. The non-determining parameter receives more arguments than the determining parameter

For example, for test 6.1 we have used as a command:

```
temples test6.templ
```

As the reader can see, there were no instantiation arguments provided. Another example, test 6.11:

```
temples test6.templ -een een twee drie vier -twee vijf zes zeven acht
```

Let us see what the results of these tests are. Test 6.1 does not have an evaluation as a result, only errors. Note that when errors are a result, then there should indeed be no evaluation, so this is not wrong. Here follows the contents of the `Terrors` result file:

```
Unknown value for dimension n
```

This is indeed the result we had expected.

Test 6.11 has as a result only the following `Tevaluation` file:

```
een twee drie vier
vijf zes zeven acht
```

Checking this with the semantics function shows that this is the correct result.

It would not be very interesting to show the results of the other tests, as the input and output are similar to the above tests. We just state that `temples` has passed all the above tests.

F.3 System testing

In the system testing, we have to make sure that the files used in the communication between `xtemples` and `temples` can be interpreted correctly by both programs. So, we have to make sure that any `Tresource`-file that is generated by `temples` can be interpreted by `xtemples` and that each instantiation call of `xtemples` does indeed map the arguments dialog correctly to the instantiation language needed by `temples`.

Testing this is impossible. Instead, we just verified whether generated file formats were usable by the reader of these files. This is indeed the case.

Bibliography

[1] Lex Augusteijn, Paul Jansen, and Harm Munk. *The Elegant Compiler Generator Tool Set.* Philips Research Laboratories, Eindhoven, release 6.2 edition.

[2] N. Barkakati. *X Window System Programming.* SAMS, first edition, 1991.

[3] A. de Bunje. *User's Manual of COLD-1.* Philips Research, Information and Software Technology, Eindhoven, 1992.

[4] L.M.G. Feijs. *A Formalisation of Design Methods.* Ellis Horwood, first edition, 1993.

[5] L.M.G. Feijs and H.B.M. Jonkers. *Specification and Design with COLD-K.* Philips Electronics, 1991.

[6] L.M.G. Feijs, H.B.M. Jonkers, C.P.J. Koymans, and G.R. Renardel de Lavalette. Formal definition of the design language COLD-K. Technical report, ESPRIT project 432, 1987.

[7] Open Software Foundation. *OSF/Motif Programmer's Guide.* Prentice Hall, release 1.2 edition, 1990.

[8] Paul Jansen. *An Introduction to Elegant.* Philips Research Laboratories, Eindhoven, first edition, 1993.

[9] H.B.M. Jonkers. Description of SPECICOLD 1.0. Technical Report RWB-508-re-92271, Philips Research, Information and Software Technology, 1992.

[10] H.B.M. Jonkers. Specifying reusable components : Guidelines and recommendations. Technical Report RWB-508-re-92012, Philips Research, Information and Software Technology, 1992.

[11] H.B.M. Jonkers. An overview of the SPRINT method. In J.C.P. Woodcock and P.E. Larsen, editors, *FME '93 : Industrial Strength Formal Methods*, pages 403–427. Springer-Verlag, 1993.

[12] Stephen M. McMenamin and John F. Palmer. *Essential Systems Analysis.* Yourdon Inc., 1984.

[13] C. Pecheur. Vlib : Infinite virtual libraries for LOTOS. In *Protocol Specification, Testing and Verification*, pages A1–1–A1–16. IFIP, 1993.

[14] Philips Research, Information and Software Technology, Eindhoven. *GENIE User's Manual*, 1993.

[15] Philips Research, Information and Software Technology, Eindhoven. *ICE (Integrated Cold Environment) 1.3*, 1993.

[16] D.A. Schmidt. *Denotational Semantics.* Allyn and Bacon, first edition, 1986.

List of Tables

List of Figures