The Common Framework Initiative
for algebraic specification and development

Michel Bidoit and Peter D. Mosses

CASL User Manual

— FIRST PUBLIC DRAFT

September 17, 2003

Springer
Berlin Heidelberg New York
Hong Kong London
Milan Paris Tokyo

Preface

CASL, the Common Algebraic Specification Language, has been designed by COFI, the Common Framework Initiative for algebraic specification and development. It is an expressive language for specifying requirements and design for conventional software. It is algebraic in the sense that models of CASL specifications are algebras; the axioms can be arbitrary first-order formulae.

From CASL, simpler languages (e.g., with axioms restricted to conditional equations, for interfacing with existing tools) are obtained by restriction, and CASL is also extended in more advanced languages (e.g., higher-order). CASL strikes a happy balance between simplicity and expressiveness.

This User Manual illustrates and discusses how to write CASL *specifications.*

Content

This CASL *User Manual* introduces the potential user to the features of CASL mainly by means of illustrative examples. It presents and discusses the typical ways in which the language concepts and constructs are expected to be used in the course of building system specifications. Thus, the presentation focuses on *what* the constructs and concepts of CASL are *for*, and *how* they should (and should not) be used. We try to make these points as clear as possible by referring to simple examples, and by discussing both the general ideas and some details of CASL specifications.

We hope that this will give the reader a sufficient feel of the formalism, and enough understanding, to look through the presentation of a basic library of CASL specifications in App. B, to follow the case study in App. C, and to

start experimenting with the use of CASL for writing specifications – perhaps employing the support tools presented in Chap. 10.

By no means, however, should this User Manual be regarded as a complete presentation of the CASL specification formalism – this is given in the accompanying volume, the CASL *Reference Manual* [Mos03].

Structure

Chapter 1 describes the origins of CASL: how CoFI was formed in response to the proliferation of algebraic specification languages in the preceding two decades, and the aims and scope that were formulated for this international initiative.

For the benefit of readers not already familiar with other algebraic specification languages, Chap. 2 will review the main concepts of algebraic specification, explaining standard terminology regarding specification language constructs and models (i.e., algebras).

Chapter 3 shows how total, many-sorted specifications in CASL may be written essentially as in many other algebraic specification languages. Loose, generated, and free specifications are discussed in turn, with illustrative examples and advice on the use of the different specification styles.

Partial functions arise naturally. Chapter 4 explains how CASL supports specification of partial functions, drawing attention to where special care is needed compared to specifications involving only total functions.

Subsorts and supersorts are often useful in CASL specifications. Chapter 5 illustrates how they can be declared and defined, and that they can sometimes avoid the need for partial functions.

The examples given so far make use of named and structured specifications in a simple and natural way. Chapter 6 takes a much closer look at the constructs CASL provides for structuring specifications, explaining how large and complex specifications are easily built out of simpler ones by means of a small number of specification-building operations.

Chapter 7 shows how making a specification generic (when appropriate) improves its reusability. It also introduces the constructs for expressing so-called views between specifications, and explains their relationship to proof obligations.

While specification-building operations are useful to structure the text of large specifications, architectural specifications are meant for imposing structure on models. Chapter 8 will discuss and illustrate the role of architectural specifications, and show how to express them in CASL.

Chapter 9 will explain and illustrate how libraries of named specifications can be formed, and made available over the Internet, to encourage widespread reuse and evolution of specifications. Version control is of crucial importance here.

Tool support is vital for efficient use of formal specifications in connection with practical software design and development. Chapter 10 presents the main tools that have been implemented so far; several of them allow use of CASL specifications in connection with tools that were originally developed for other specification languages, showing how CASL provides tool interoperability.

Chapter 11 gives a detailed overview of the foundations of CASL, which are established in the accompanying CASL *Reference Manual* [Mos03].

This User Manual is completed by three appendices: App. A provides an compact overview of all CASL constructs, for quick reference; App. B introduces a few of the many specifications that are available in the CASL libraries of basic datatypes at Bremen; and finally, App. C will give a realistic case-study of the use of CASL in practice, in connection with the design of software for a Steam-Boiler.

Organization

All the main points are highlighted like this.

The material in the User Manual is organized in a tutorial fashion.[1] Each point is usually accompanied by an illustrative example of a complete CASL specification; the names of these specifications are listed (both in order of presentation and alphabetically) at the end of the book.

Readers who are familiar with previous algebraic specification languages, and especially those who have been following or participating in the design and development of CASL, may prefer to skip lightly through Chaps. 1 and 2. Chapter 3, however, is mandatory, since it is there that we introduce many CASL features that will be needed to understand the subsequent chapters.

In contrast, Chaps. 4 and 5 can be skipped at first reading if the reader is not so much interested in these features (with the proviso though that there are some references to the examples given in these chapters from later chapters).

Chapters 6 and 7 present mainstream material, and until one feels comfortable with all the main points and examples, it is advisable to wait with proceeding to Chap. 8. Chapter 9 is however quite orthogonal to the other chapters, and can be read when desired. Part of the introduction to CASL support tools in Chap. 10 assumes familiarity with concepts introduced in Chap. 7. Chapter 11 is primarily for those who will want to follow up on this User Manual with a more detailed study of the language, based on the Reference Manual.

[1] It should be possible to use it as the basis for an interactive tutorial, but this is not yet available.

Acknowledgements

Chapter 10 was written by Till Mossakowski, who also provided App. B, based on joint work with Markus Roggenbach and Lutz Schröder. Chapter 11 was written by Donald Sannella and Andrzej Tarlecki. Comments on previous drafts of Chaps. 3–6 from CoFI participants have been very helpful.

Draft Status

Many of the main chapters are close to a final draft. However, some of the less central chapters (Chaps. 1, 2, and 9) have been given lower priority, and are still at the outline stage. A full draft of Chapter 8 is expected to become available by the beginning of August, followed by Appendix C.

We look forward to all comments based on the present draft of the User Manual, either addressed personally to the authors, or (preferably) submitted to the (moderated) CoFI discussion mailing list, cofi-discuss@cofi.info.

Michel Bidoit and Peter D. Mosses
15 July, 2003

Contents

1

Introduction

1.1 CoFI

CoFI *is an initiative to provide a common framework for algebraic specification and development.*

- *Background*
- *Aims*
- *Status*

1.2 Casl

Casl *has been designed by* CoFI *as a general-purpose algebraic specification language, subsuming many existing languages.*

- *General features*
- *Major parts*
- *Specific features*

2

Underlying Concepts

2.1 Specifications

- *Symbols*
- *Declarations*
- *Axioms*
- *Structure*
- *Architecture*
- *Libraries*

2.2 Algebras

- *Signatures*
- *Models*
- *Classes of models*

3

Total, Many-Sorted Specifications

Total, many-sorted specifications in CASL *may be written essentially as in many other algebraic specification languages.*

The simplest kind of algebraic specification is when each specified operation is to be interpreted as an ordinary *total* mathematical function: it takes values of particular types as arguments, and always returns a well-defined value. Total functions correspond to software whose execution always terminates normally. The types of values are distinguished by simple symbols called *sorts*.

In practice, a realistic software specification involves *partial* as well as total functions. However, it may well be formed from simpler specifications, some of which involve only total functions. This chapter explains how to express such total specifications in CASL, illustrating various features of CASL.

Total many-sorted specifications can also be expressed in many previous specification languages; it is usually straightforward to reformulate them in CASL. Readers who know other specification languages will probably recognize some familiar examples among the illustrations given in this chapter.

CASL *provides also useful abbreviations.*

The technique of algebraic specification by axioms is generally well-suited to expressing properties of functions. However, when functions have commonly-occurring mathematical properties, it can be tedious to give the corresponding axioms explicitly. CASL provides some useful abbreviations for such cases. For instance, so-called free datatype declarations allow sorts and value constructors to be specified much as in functional programming languages, using a concise and suggestive notation.

CASL *allows loose, generated and free specifications.*

The models of a loose specification include all those where the declared functions have the specified properties, without any restrictions on the sets of values corresponding to the various sorts. In models of a generated specification, in contrast, it is required that all values can be expressed by terms formed from the specified functions, i.e. unreachable values are prohibited. In models of free specifications, it is required that values of terms are distinct except when their equality follows from the specified axioms: the possibility of unintended coincidence between them is prohibited.

Section 3.1 below focusses on loose specifications; Sect. 3.2 discusses the use of generated specifications, and Sect. 3.3 does the same for free specifications. Loose, generated, and free specifications are often used together in CASL: each style has its advantages in particular circumstances, as explained below in connection with the illustrative examples.

3.1 Loose Specifications

CASL *syntax for declarations and axioms involves familiar notation, and is mostly self-explanatory.*

spec STRICT_PARTIAL_ORDER =
%% Let's start with a simple example !
sort *Elem*
pred $\_\_ < \_\_ : Elem \times Elem$ %% **pred** abbreviates predicate
$\forall x, y, z : Elem$
- $\neg(x < x)$ %(strict)%
- $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
- $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

%{ Note that there may exist x, y such that neither $x < y$ nor $y < x$. }%
end

The above (basic) specification, named STRICT_PARTIAL_ORDER, introduces a sort *Elem* and a binary infix predicate symbol '<'. Note that CASL allows so-called *mixfix notation*,[1] i.e., the specifier is free to indicate, using '__' (pairs of underscores) as *place-holders*, how to place arguments when building terms (single underscores are treated as letters in identifiers). Using mixfix

[1] Mixfix notation is so-called because it generalizes infix, prefix, and postfix notation to allow arbitrary mixing of argument positions and identifier tokens.

notation generally allows the use of familiar mathematical and programming notations, which contributes substantially to the readability of specifications.

The interpretation of the binary predicate symbol '$<$' is then constrained by three axioms. A set of axioms is generally presented as a 'bulleted' list of formulae, preceded by the universally quantified declaration of the relevant variables, together with their respective sorts, as shown in the above example. In CASL, axioms are written in first-order logic with equality, using quantifiers and the usual logical connectives. The universal quantification preceding a list of axioms applies to the entire list. Axioms can be annotated by labels written %(...)%, which is convenient for proper reference in explanations or by tools.

Note that '$\forall$' is input as '`forall`', and that '$\bullet$' is input as '`.`' or '$\cdot$'. The usual logical connectives '$\Rightarrow$', '$\Leftrightarrow$', '$\wedge$', '$\vee$', and '$\neg$', are input as '`=>`', '`<=>`', '`/\`', '`\/`', and '`not`', respectively; '$\neg$' can also be input directly as an ISO Latin-1 character. The existential quantifier '$\exists$' is input as '`exists`'.

It is advisable to comment as appropriate the various elements introduced in a specification. The syntax for end-of-line and grouped multi-line comments is illustrated in the above example.

The above STRICT_PARTIAL_ORDER specification is loose in the sense it has many (non-isomorphic) models, among which models where '$<$' is interpreted by a total ordering relation and models where it is interpreted by a partial one.

Specifications can easily be extended by new declarations and axioms.

spec TOTAL_ORDER =
STRICT_PARTIAL_ORDER
then $\forall x, y : Elem \bullet x < y \vee y < x \vee x = y$ %(total)%
end

Extensions, introduced by the keyword '**then**', may specify new symbols, possibly constrained by some axioms, or merely require further properties of old ones, as in the above TOTAL_ORDER example, or more generally do both at the same time. In TOTAL_ORDER, we further constrain the interpretation of the predicate symbol '$<$' by requiring it to be a total ordering relation.

All symbols introduced in a specification are by default exported by it and visible in its extensions. This is for instance the case here for the sort *Elem* and the predicate symbol '$<$', which are introduced in STRICT_PARTIAL_ORDER, exported by it, and therefore available in TOTAL_ORDER.[2]

[2] See Chap. 6 for constructs allowing the explicit restriction of the set of symbols exported by a specification.

In simple cases, an operation (or a predicate) symbol may be declared and its intended interpretation defined at the same time.

spec TOTAL_ORDER_WITH_MINMAX =
TOTAL_ORDER
then **ops** $min(x, y : Elem) : Elem = x \ when \ x < y \ else \ y$;
$max(x, y : Elem) : Elem = y \ when \ min(x, y) = x \ else \ x$
end

TOTAL_ORDER_WITH_MINMAX extends TOTAL_ORDER by introducing two binary operation symbols *min* and *max*, for which a functional notation is to be used, so no place-holders are given. The intended interpretation of the symbol *min* is defined simultaneously with its declaration, and the same is done for *max*. For instance,

op $min(x, y : Elem) : Elem = x \ when \ x < y \ else \ y$

abbreviates:

op $min : Elem \times Elem \to Elem$
$\forall x, y : Elem \bullet min(x, y) = x \ when \ x < y \ else \ y$

(and similarly for *max*).

The '... *when* ... *else* ...' construct used above is itself an abbreviation. Indeed, $min(x, y) = x \ when \ x < y \ else \ y$ abbreviates $(x < y \Rightarrow min(x, y) = x) \wedge (\neg(x < y) \Rightarrow min(x, y) = y)$.

In CASL specifications, visibility is *linear*, i.e., any symbol must be declared before being used. In the above example, *min* should be declared before being used to define *max*.

Linear visibility does not imply, however, that a fixed scheme is to be used when writing specifications: the specifier is free to present the required declarations and axioms in any order, as long as the linear visibility rule is respected. For instance, one may prefer to declare first all sorts and all operation or predicate symbols needed, and then specify their properties by the relevant axioms. Or, in contrast, one may prefer to have each operation or predicate symbol declaration immediately followed by the axioms constraining its interpretations. Both styles are equally fine, and can even be mixed if desired. This flexibility is illustrated in the following variant of the TOTAL_ORDER_WITH_MINMAX specification, where for explanatory purposes we refrain from using the useful abbreviations explained above.

spec VARIANT_OF_TOTAL_ORDER_WITH_MINMAX =
TOTAL_ORDER
then **vars** $x, y : Elem$
op $min : Elem \times Elem \to Elem$
• $x < y \Rightarrow min(x, y) = x$
• $\neg(x < y) \Rightarrow min(x, y) = y$

op $max : Elem \times Elem \to Elem$
- $x < y \Rightarrow max(x, y) = y$
- $\neg(x < y) \Rightarrow max(x, y) = x$

end

Note that in order to avoid the tedious repetition of the declaration of the variables x and y for each list of axioms, we have used here a *global* variable declaration which introduces x and y for the rest of the specification. Variable declarations are of course not exported across specification extensions: the variables x and y declared in VARIANT_OF_TOTAL_ORDER_WITH_MINMAX are not visible in any of its extensions.

Symbols may be conveniently displayed as usual mathematical symbols by means of ***%display*** *annotations.*

%display __`leq`__ %LATEX __ $\leq$ __
spec PARTIAL_ORDER =
  STRICT_PARTIAL_ORDER
then **pred** __ $\leq$ __$(x, y : Elem) \Leftrightarrow (x < y \vee x = y)$
end

The above example relies on a **%display** annotation: while, for obvious reasons, the specification text should be *input* using the ISO Latin-1 character set, it is often convenient to *display* some symbols differently, e.g., as mathematical symbols. This is the case here where the '`leq`' predicate symbol is more conveniently displayed as '$\leq$'. Display annotations, as any other CASL annotations, are auxiliary parts of a specification, for use by tools, and do not affect the semantics of the specification.[3]

In the above example, we have again used the facility of simultaneously declaring and defining a symbol (here, the predicate symbol '$\leq$') in order to obtain a more concise specification.

The ***%implies*** *annotation is used to indicate that some axioms are supposed to be consequences of others.*

spec PARTIAL_ORDER_1 =
  PARTIAL_ORDER
then %implies
  $\forall x, y, z : Elem \bullet x \leq y \wedge y \leq z \Rightarrow x \leq z$ %(transitive)%
end

[3] Display annotations should be provided at the beginning of a library, see Chap. 9.

The **%implies** annotation above is used to emphasize that the transitivity of '$\leq$' should follow from the other axioms, or, in other words, that the model class of PARTIAL_ORDER_1 is exactly the same as the model class of PARTIAL_ORDER.

Note however that an annotation does not affect the semantics of a specification (hence removing the **%implies** annotation does not change the semantics of the above specification), but merely provides useful proof obligations. The sole aim of an **%implies** annotation is to stress the specifier's intentions, and it will also help readers confirm their understanding. Some tools may of course use such annotations to generate corresponding proof obligations (here, the proof obligation is PARTIAL_ORDER $\models \forall x, y, z : Elem \bullet x \leq y \wedge y \leq z \Rightarrow x \leq z$). Discharging these proof obligations increases the trustworthiness of a specification.

To fully understand that an **%implies** annotation has no effect on the semantics, the best is to consider an example where the corresponding proof obligation cannot be discharged, as shown below.

spec IMPLIES_DOES_NOT_HOLD =
PARTIAL_ORDER
then %implies
$\forall x, y : Elem \bullet x < y \vee y < x \vee x = y$ %(total)%
end

Since the loose specification PARTIAL_ORDER has models where '$<$' is interpreted by a partial ordering relation, the proof obligation corresponding to the above **%implies** annotation cannot be discharged. However, since annotations have no impact on the semantics, the specification IMPLIES_DOES_NOT_HOLD is not inconsistent, it just constrains the interpretation of '$<$' to be a total ordering relation.

Attributes may be used to abbreviate axioms for associativity, commutativity, idempotency, and unit properties.

spec MONOID =
sort *Monoid*
ops *1* : *Monoid*;
$\_\_ * \_\_ : Monoid \times Monoid \rightarrow Monoid$, *assoc*, *unit 1*
end

The above example introduces a constant symbol *1* of sort *Monoid*, then a binary operation symbol '$*$', which is asserted to be associative and to have *1* as unit element.

The *assoc* attribute abbreviates, as expected, the following axiom:
$\forall x, y, z : Monoid \bullet (x * y) * z = x * (y * z)$
The '*unit 1*' attribute abbreviates:
$\forall x : Monoid \bullet (x * 1 = x) \wedge (1 * x = x)$

Note that to make the use of '*unit 1*' legal, it is necessary to have previously declared the constant *1*, to respect the linear visibility rule.

Other available attributes are *comm*, which abbreviates the obvious axiom stating that a binary operation is commutative, and *idem*, which can be used to assert the idempotency of a binary operation f (i.e., that $f(x, x) = x$).

Asserting '$*$' to be associative using the attribute *assoc* makes the term $x * y * z$ well-formed (assuming x, y, z of the right sort), while otherwise grouping parentheses would be required. Moreover, it is expected that some tools (e.g., systems based on rewriting) may use the *assoc* attribute, so it is generally advisable to use this attribute instead of stating the same property by an axiom (the same applies to the other attributes).

Genericity of specifications can be made explicit using parameters.

spec GENERIC_MONOID [**sort** *Elem*] =
  sort *Monoid*
  ops *inj* : *Elem* → *Monoid*;
    1 : *Monoid*;
    __ * __ : *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
  ∀ *x*, *y* : *Elem* • *inj*(*x*) = *inj*(*y*) ⇒ *x* = *y*
end

The above example describes monoids built over arbitrary elements (of sort *Elem*). The intention here is to reuse the specification GENERIC_MONOID to derive from it specifications of monoids built over, say, characters, symbols, etc. In such cases, it is appropriate to emphasize the intended genericity of the specification by making explicit, in a distinguished *parameter part* (here, [**sort** *Elem*]), the piece of specification that is intended to vary in the derived specifications. In these, it will then be possible to *instantiate* the parameter part as desired in order to specialize the specification as appropriate (to obtain, e.g., a specification of monoids built over characters). A named specification with one or more parameter(s) is called *generic*.

The body of the generic specification GENERIC_MONOID is an extension of what is specified in the parameter part. Hence an alternative to the above generic specification GENERIC_MONOID is the following, less elegant, non-generic specification (which cannot be specialized by instantiation):

spec NON_GENERIC_MONOID =
  sort *Elem*
then **sort** *Monoid*
  ops *inj* : *Elem* → *Monoid*;
    1 : *Monoid*;
    __ * __ : *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
  ∀ *x*, *y* : *Elem* • *inj*(*x*) = *inj*(*y*) ⇒ *x* = *y*
end

A generic specification may have more than one parameter, and parameters can be arbitrary specifications, named or not. When reused by reference to its name, a generic specification *must* be instantiated. Generic specifications and how to instantiate them are discussed in detail later in Chap. 7. Using generic specifications when appropriate improves the reusability of specification definitions.

References to generic specifications always instantiate the parameters.

spec GENERIC_COMMUTATIVE_MONOID [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then $\forall x, y : Monoid \bullet x * y = y * x$
end

The above (generic) specification GENERIC_COMMUTATIVE_MONOID is defined as an extension of GENERIC_MONOID, which should therefore be instantiated, as explained above. Instantiating a generic specification is done by providing an argument specification that 'fits' the parameter part of the generic specification to be instantiated.

It is however quite frequent that the instantiation is 'trivial', i.e., the argument specification is identical to the parameter one. This is the case for the above example, where the generic specification GENERIC_MONOID is instantiated by providing the same argument specification '**sort** *Elem*' as the original parameter.

spec GENERIC_COMMUTATIVE_MONOID_1 [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then **op** $\_\_ * \_\_ : Monoid \times Monoid \to Monoid, comm$
end

GENERIC_COMMUTATIVE_MONOID_1 is an alternative version of the former specification where, instead of requiring explicitly with an axiom the commutativity of the operation '$*$', we require it using the attribute *comm*. As explained before, it is in general better to describe such requirements using attributes rather than explicit axioms, since it is expected that some tools will rely on these attributes for specialized algorithms (e.g., ACI-unification).

This example illustrates also an important feature of CASL, the *'same name, same thing'* principle. The operation symbol '$*$' is indeed declared twice, with the same profile, first in GENERIC_MONOID and then again in GENERIC_COMMUTATIVE_MONOID_1 (the second declaration being enriched by the attribute *comm*). This is perfectly fine and defines only one binary operation symbol '$*$' with the corresponding arity, according to the 'same name, same thing' principle. This principle applies to sorts, as well as to operation and predicate symbols. It applies both to symbols defined locally and to symbols imported from an extended specification, as it is the case here for '$*$'. Of course, it does not apply between named specifications, i.e.,

the same symbol may be used in different named specifications with entirely different interpretations.

Note that for operation and predicate symbols, the 'same name, same thing' principle is a little more subtle than for sorts: the 'name' of an operation (or of a predicate) includes its profile of argument and result sorts, so two operations defined with the same symbol but with different profiles do not have the same 'name', the symbol is just *overloaded*. When an overloaded symbol is used, the intended profile is to be determined by the context (e.g., the sorts of the arguments to which the symbol is applied). Explicit disambiguation can be used when needed, by specifying the profile (or result sort) in an application.[4] Note that overloaded constants are allowed in CASL (e.g., *empty* may be declared to be a constant of various sorts of collections).

Datatype declarations may be used to abbreviate declarations of sorts and constructors.

spec CONTAINER [**sort** *Elem*] =
  type *Container* ::= *empty* | *insert*(*Elem*; *Container*)
  pred *__is_in__* : *Elem* × *Container*
  ∀ *e*, *e*′ : *Elem*; *C* : *Container*
  • ¬(*e is_in empty*)
  • *e is_in insert*(*e*′, *C*) ⇔ (*e* = *e*′ ∨ *e is_in C*)
end

Specifications of 'datatypes' with constructors are frequently needed. CASL provides special constructs for datatype declarations to abbreviate the corresponding rather tedious declarations. For instance, the above datatype declaration:

  type *Container* ::= *empty* | *insert*(*Elem*; *Container*)

abbreviates:

  sort *Container*
  ops *empty* : *Container*;
    insert : *Elem* × *Container* → *Container*

A datatype declaration looks like a context-free grammar in a variant of BNF. It declares the symbols on the left of '::=' as sorts, and for each alternative on the right it declares a constructor.

A datatype declaration as the one above is *loose* since it does not imply any constraint on the values of the declared sorts: there may be some values of sort *Container* that are not built by any of the declared constructors, and the same value may be built by different applications of the constructors to some arguments.

[4] For instance, depending on the context, the term *t1* ∗ *t2* can be disambiguated by writing (*op* ∗ : *Monoid* × *Monoid* → *Monoid*)(*t1*, *t2*), or just (*t1* : *Monoid*) ∗ (*t2* : *Monoid*), or even (*t1* ∗ *t2*) : *Monoid*.

Datatype declarations may also be specified as generated (see Sect. 3.2) or free (see Sect. 3.3). Moreover, selectors, which are usually partial operations, may be specified for each component (see Chap. 4).

As a last minor remark, note that while placeholders are specified with pairs of underscores '__', a single underscore is a valid character in an identifier, as illustrated in the above specification by the predicate symbol *is_in*.

> *Loose datatype declarations are appropriate when further constructors may be added in extensions.*

spec MARKING_CONTAINER [**sort** *Elem*] =
CONTAINER [**sort** *Elem*]
then **type** $Container ::= mark\_insert(Elem;\ Container)$
pred $\_\_is\_marked\_in\_\_ : Elem \times Container$
$\forall e, e' : Elem;\ C : Container$
- $e\ is\_in\ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$
- $\neg(e\ is\_marked\_in\ empty)$
- $e\ is\_marked\_in\ insert(e', C) \Leftrightarrow e\ is\_marked\_in\ C$
- $e\ is\_marked\_in\ mark\_insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_marked\_in\ C)$

end

The above specification extends CONTAINER (trivially instantiated) by introducing another constructor *mark_insert* for the sort *Container* (hence, values added to a container may now be 'marked' or not). Note that we heavily rely on the 'same name, same thing' principle here, since it ensures that the sort *Container* introduced by the datatype declaration of CONTAINER and the sort *Container* introduced by the datatype declaration of MARKING_CONTAINER are the same sort, which implies that the combination of both datatype declarations is equivalent to:

type $Container ::= empty \mid insert(Elem;\ Container) \mid mark\_insert(Elem;\ Container)$

Note that since 'new' values may be constructed by *mark_insert*, it is necessary to extend the specification of the predicate symbol *is_in* by an extra axiom taking care of the newly introduced constructor.

3.2 Generated Specifications

Sorts may be specified as generated by their constructors.

spec GENERATED_CONTAINER [**sort** *Elem*] =
generated type *Container* ::= *empty* | *insert*(*Elem*; *Container*)
pred __*is_in*__ : *Elem* × *Container*
∀ e, e' : *Elem*; *C* : *Container*
• ¬(*e is_in empty*)
• *e is_in insert*(e', C) ⇔ ($e = e' \vee e$ *is_in C*)
end

When a datatype is declared as *generated*, as in the above example, the corresponding sort is constrained to be generated by the declared constructors, which means that any value of this sort is built by some constructor application. This constraint is sometimes referred to as the *'no junk'* principle. For instance, in the above example, having declared the datatype *Container* to be generated entails that in any model of GENERATED_CONTAINER, any value of sort *Container* is denotable by a term built with *empty*, *insert*, and variables of sort *Elem* only.

As a consequence, properties of values of sort *Container* can be proved by induction on the declared constructors. A major benefit of generated datatypes is indeed that induction on the declared constructors is a sound proof principle.

The construct '**generated type** ...' used above is just an abbreviation for '**generated { type** ... **}**', and '**generated**' can be used around arbitrary signature declarations, enclosed in '**{**' and '**}**'.

Generated specifications are in general loose.

spec GENERATED_CONTAINER_MERGE [**sort** *Elem*] =
GENERATED_CONTAINER [**sort** *Elem*]
then op __*merge*__ : *Container* × *Container* → *Container*
∀ *e* : *Elem*; *C*, C' : *Container*
• *e is_in* (*C merge* C') ⇔ (*e is_in C* ∨ *e is_in* C')
end

A generated specification is in general loose. For instance, GENERATED_CONTAINER is loose since, although all values of sort *Container* are specified to be generated by *empty* and *insert*, the behaviour of the *insert* constructor is still loosely specified (nothing is said about the case where an element is inserted to a container which already contains this element). Hence GENERATED_CONTAINER admits several non-isomorphic models.

GENERATED_CONTAINER_MERGE is as loose as GENERATED_CONTAINER in what concerns *insert*, and in addition, the newly introduced operation symbol *merge* is also loosely specified: nothing is said about what happens when merging two containers which share some elements.

It is important to understand that looseness of a specification is not a problem, but on the contrary avoids unnecessary overspecification. In particular, loose specifications are in general well-suited to capture requirements.

The fact that *merge* is loosely specified does not mean that it can produce new values of the sort *Container*. On the contrary, since this sort has been specified as being generated by *empty* and *insert*, it follows that any value denotable by a term of the form $merge(\ldots,\ldots)$ can also be denoted by a term built with *empty* and *insert* (and no *merge*). Hence, for the specification GENERATED_CONTAINER_MERGE, proofs by induction on *Container* only need to consider *empty* and *insert*, and not *merge*, as was the case for GENERATED_CONTAINER.

Generated specifications need not be loose.

spec GENERATED_SET [**sort** *Elem*] =
  generated type $Set ::= empty \mid insert(Elem;\ Set)$
  pred $\_\_is\_in\_\_ : Elem \times Set$
  ops $\{\_\_\}(e : Elem) : Set = insert(e, empty)$;
    $\_\_ \cup \_\_ : Set \times Set \to Set$;
    $remove : Elem \times Set \to Set$
  $\forall e, e' : Elem;\ S, S' : Set$
  - $\neg(e\ is\_in\ empty)$
  - $e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$
  - $S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$ %(equal_sets)%
  - $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
  - $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$

then %implies
  $\forall e, e' : Elem;\ S : Set$
  - $insert(e, insert(e, S)) = insert(e, S)$
  - $insert(e, insert(e', S)) = insert(e', insert(e, S))$

  generated type $Set ::= empty \mid \{\_\_\}(Elem) \mid \_\_ \cup \_\_(Set;\ Set)$
  op $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty$
end

Although generated specifications are in general loose, they need not be so, as illustrated by the above GENERATED_SET specification, where the axiom %(equal_sets)%, combined with the axioms defining *is_in*, fully constrains the interpretations of the sort *Set* and of the constructors *empty* and *insert*, once an interpretation for the sort *Elem* is chosen.

Note also that this example displays the power of the annotation **%implies**, which is used here not only to stress that the usual properties of *insert* are expected to follow from the preceding declarations and axioms, but also that an alternative induction scheme, based on *empty*, $\{\_\_\}$ and $\_\_ \cup \_\_$, can be used for sets. Moreover, it asserts that $\_\_ \cup \_\_$ is expected to be associative, commutative, idempotent (i.e., $S \cup S = S$), and to have *empty* for unit. Note again that this **%implies** part heavily relies on the 'same name, same meaning' principle.

Generated types may need to be declared together.

The following specification fragment illustrates what may go wrong:

sort *Node*
generated type *Tree* ::= *mktree*(*Node*; *Forest*)
generated type *Forest* ::= *empty* | *add*(*Tree*; *Forest*)

The above is both *incorrect*, due to the linear visibility rule,[5] and *wrong*, since the corresponding semantics is not what a naive reader may expect. One may expect that only the minimal fixpoint interpretation of the above mutually recursive definitions of the datatypes *Tree* and *Forest* is acceptable, but indeed any fixpoint interpretation is, which means that, for instance, a model with both a junk tree (obtained from a node and a junk forest by *mktree*) and a junk forest (obtained from a junk tree and an empty forest by *add*) fulfills the above declarations. Hence, one must write instead:

sort *Node*
generated types *Tree* ::= *mktree*(*Node*; *Forest*);
Forest ::= *empty* | *add*(*Tree*; *Forest*)

Here, the mutually recursive datatypes *Tree* and *Forest* are correctly defined simultaneously within the same **generated types** construct, and the resulting semantics is the expected one (without junk values for trees and forests). Note that therefore, the linear visibility rule is *not* applicable *within* a **generated types** construct (to allow such mutually recursive definitions), but that this is the only exception to the linear visibility principle. Only mutually recursive generated datatypes need to be declared together; in simpler cases, it makes no difference to have a sequence of successive generated datatype declarations or just one introducing all the desired datatypes.[6]

[5] This can easily be fixed by replacing '**sort** *Node*' by '**sorts** *Node*, *Tree*, *Forest*'.
[6] The same explanations apply to free datatypes, introduced in the next subsection.

3.3 Free Specifications

Free specifications provide initial semantics and avoid the need for explicit negation.

spec NATURAL = **free type** $Nat ::= 0 \mid suc(Nat)$ **end**

A **free** datatype declaration corresponds to the so-called '*no junk, no confusion*' principle: there are no other values of sort *Nat* than those denoted by the constructor terms built with *0* and *suc*, and all distinct constructor terms denote different values.

Hence, a **free** datatype declaration as the one above has *exactly* the same effect as the similar **generated** datatype declaration, *together* with axioms stating that *suc* is injective, and that *0* cannot be the successor of a natural number. An alternative to the above '**free type** $Nat ::= 0 \mid suc(Nat)$' is therefore:

generated type $Nat ::= 0 \mid suc(Nat)$
$\forall x, y : Nat \bullet suc(x) = suc(y) \Rightarrow x = y$
$\forall x : Nat \bullet \neg(0 = suc(x))$

Free datatype declarations are particularly convenient for defining enumerated datatypes.

spec COLOUR =
free type $RGB ::= Red \mid Green \mid Blue$
free type $CMYK ::= Cyan \mid Magenta \mid Yellow \mid Black$
end

Using '**free**' instead of '**generated**' for defining enumerated datatypes saves the writing of many explicit distinctness assertions (e.g., $\neg(Red = Green)$, $\neg(Red = Blue)$, ...).

Free specifications can also be used when the constructors are related by some axioms.

spec INTEGER =
free { **type** $Int ::= 0 \mid suc(Int) \mid pre(Int)$
$\forall x, y : Int$
$\bullet\ suc(pre(x)) = x$
$\bullet\ pre(suc(x)) = x$ **}**
end

When some relations are to be imposed between the constructors (as is the case here for *suc* and *pre* which are inverses of each other), a **free** datatype declaration cannot be used, since the contradiction between the 'no confusion' principle and the axioms imposed on the constructors would lead to an inconsistent specification. Instead, one should impose a '*freeness constraint*' around the datatype declaration followed by the required axioms. A freeness constraint, expressed by the keyword **free**, can be imposed around any specification.

In the case of the above INTEGER specification, the freeness constraint imposes that the semantics of the specification is the quotient of the constructor terms by (the minimal congruence induced by) the given axioms, and hence provides exactly the desired semantics. More generally, a freeness constraint around a specification indicates its initial model, which may of course not exist. It is however well-known that initial models of basic specifications with axioms restricted to Horn clauses always exist. Remember also that equality holds minimally in initial models of equational specifications.

Predicates hold minimally in models of free specifications.

spec NATURAL_ORDER =
NATURAL
then **free** { **pred** $\_\_ < \_\_ : Nat \times Nat$
$\forall x, y : Nat$
• $0 < suc(x)$
• $x < y \Rightarrow suc(x) < suc(y)$ }
end

A freeness constraint imposed around a predicate declaration followed by some defining axioms has the effect that the predicate only holds when this follows from the given axioms, and does not hold otherwise. For instance, in the above example, it is not necessary to explicitly state that '$\neg(0 < 0)$', since this will follow from the imposed freeness constraint. Hence, in such cases a freeness constraint has exactly the same effect as the so-called '*negation by default*' or '*closed world assumption*' principles in logic programming. More generally, it is often convenient to define a predicate within a freeness constraint, since by doing so, one has to specify the 'positive' cases only.

Operations may be safely defined by induction on the constructors of a free datatype declaration.

spec NATURAL_ARITHMETIC =
NATURAL_ORDER
then **ops** $1 : Nat = suc(0)$;
$\_\_ + \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 0$;
$\_\_ * \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 1$
$\forall x, y : Nat$
- $x + suc(y) = suc(x + y)$
- $x * 0 = 0$
- $x * suc(y) = x + (x * y)$

end

To define some operation on a free datatype, it is generally recommended to make a case distinction with respect to the various constructors defined. This is illustrated by the above definitions of '+' and '$*$' (although for the '+' operation, the case for the constructor *0* is already taken care of by the attribute '*unit 0*').[7]

More care may be needed when defining operations on free datatypes with axioms relating the constructors.

spec INTEGER_ARITHMETIC =
INTEGER
then **ops** $1 : Int = suc(0)$;
$\_\_ + \_\_ : Int \times Int \to Int, assoc, comm, unit\ 0$;
$\_\_ - \_\_ : Int \times Int \to Int$;
$\_\_ * \_\_ : Int \times Int \to Int, assoc, comm, unit\ 1$
$\forall x, y : Int$
- $x + suc(y) = suc(x + y)$
- $x + pre(y) = pre(x + y)$
- $x - 0 = x$
- $x - suc(y) = pre(x - y)$
- $x - pre(y) = suc(x - y)$
- $x * 0 = 0$
- $x * suc(y) = (x * y) + x$
- $x * pre(y) = (x * y) - x$

end

[7] In specification libraries, ordinary decimal notation for natural numbers can be provided by use of so-called literal syntax annotations, see Chap. 9.

While a case distinction with respect to the constructors of a free datatype is harmless, this may not be the case for a datatype defined within a freeness constraint, since, due to the axioms relating the constructors to each other, some cases may overlap. This does not mean, however, that one cannot use the case distinction, but just that more attention should be paid than for a free datatype. For instance, in the above example no problem arises. But one should be more careful with the next one, since a negative integer can be of the form $suc(x)$, hence asserting, e.g., $0 \leq suc(x)$, would of course be wrong.

spec INTEGER_ARITHMETIC_ORDER =
INTEGER_ARITHMETIC
then **preds** $\_\_ \leq \_\_, \_\_ \geq \_\_, \_\_ < \_\_, \_\_ > \_\_ : Int \times Int$
$\forall x, y : Int$
- $0 \leq 0$
- $\neg(0 \leq pre(0))$
- $0 \leq x \Rightarrow 0 \leq suc(x)$
- $\neg(0 \leq x) \Rightarrow \neg(0 \leq pre(x))$
- $suc(x) \leq y \Leftrightarrow x \leq pre(y)$
- $pre(x) \leq y \Leftrightarrow x \leq suc(y)$
- $x \geq y \Leftrightarrow y \leq x$
- $x < y \Leftrightarrow (x \leq y \wedge \neg(x = y))$
- $x > y \Leftrightarrow y < x$

end

Generic specifications often involve free extensions of (loose) parameters.

spec LIST [**sort** *Elem*] =
free type $List ::= empty \mid cons(Elem; List)$
end

The parameter of a generic specification should be loose to cope with the various expected instantiations. On the other hand, it is a frequent situation that the body of the generic specification should have a free, initial interpretation. This is illustrated by the above example, where we want to combine a loose interpretation for the sort *Elem* with a free interpretation for lists. The following example is similar in spirit.

spec SET [**sort** *Elem*] =
free { **type** $Set ::= empty \mid insert(Elem; Set)$
pred $\_\_is\_in\_\_ : Elem \times Set$
$\forall e, e' : Elem; S : Set$
- $insert(e, insert(e, S)) = insert(e, S)$
- $insert(e, insert(e', S)) = insert(e', insert(e, S))$

- $\neg(e\ is\_in\ empty)$
- $e\ is\_in\ insert(e, S)$
- $e\ is\_in\ insert(e', S)\ if\ e\ is\_in\ S$ **}**

end

As for the LIST example, we want to have a loose interpretation for the sort *Elem* and a free interpretation for sets. Since some axioms are required to hold for the *Set* constructors *empty* and *insert*, we cannot use a free datatype declaration, hence we use a freeness constraint.

Note that since, as already explained, predicates hold minimally in models of free specifications, it would have been enough, in the above example, to define the predicate *is_in* by the sole axiom $e\ is\_in\ insert(e, S)$.[8] However, doing so would have decreased the comprehensibility of the specification and this is the reason why we have preferred a more verbose axiomatization of the predicate *is_in*.

Note also the use of the keyword '*if*' to write an implication in the reverse order: $e\ is\_in\ insert(e', S)\ if\ e\ is\_in\ S$ is equivalent to $e\ is\_in\ S \Rightarrow e\ is\_in\ insert(e', S)$.

The following example specifies the transitive closure of an arbitrary binary relation R on some sort *Elem* (both provided by the parameter).

spec TRANSITIVE_CLOSURE [**sort** *Elem* **pred** $\_\_R\_\_ : Elem \times Elem$] =
free { **pred** $\_\_R^+\_\_ : Elem \times Elem$
$\forall x, y, z : Elem$
- $x\ R\ y \Rightarrow x\ R^+ y$
- $x\ R^+ y \wedge y\ R^+ z \Rightarrow x\ R^+ z$ **}**

end

In the above example, it is crucial that predicates hold minimally in models of free specifications, since this property ensures that what we define as 'R^+' is actually the smallest transitive relation including R. Without requiring the freeness constraint, one would allow arbitrary relations containing the transitive closure of R (and these undesired relations cannot be eliminated merely by specifying further axioms).

Loose extensions of free specifications can avoid overspecification.

spec NATURAL_WITH_BOUND =
NATURAL_ARITHMETIC
then **op** *max_size* : *Nat* **axiom** $0 < max\_size$ **end**

The above example shows another benefit of mixing loose and initial semantics. Assume that at this stage we want to introduce some bound, of sort

[8] If an element e belongs to a set S', then this set S' can always be denoted by a constructor term of the form $insert(e, S)$, due to the axioms constraining the constructor *insert*.

Nat, without fixing its value yet (this value is likely to be fixed later in some refinements, and all that we need for now is the existence of some bound). This is provided by the above specification NATURAL_WITH_BOUND, where we mix an initial interpretation for the sort *Nat* (defined using a free datatype declaration in NATURAL) and a loose interpretation for the constant *max_size*. Each model of NATURAL_WITH_BOUND will provide a fixed interpretation of the constant *max_size*, and all these models are captured by NATURAL_WITH_BOUND, which is in this sense loose. Using such loose extensions is in general appropriate to avoid unnecessary overspecification.

spec SET_CHOOSE [**sort** *Elem*] =
SET [**sort** *Elem*]
then **op** *choose* : $Set \to Elem$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$
end

This example shows again the benefit of mixing initial and loose semantics. Here, we want to extend sets, defined using a free constraint in SET, by a loosely specified operation *choose*.[9] At this stage, the only property required for *choose* is to provide some element belonging to the set to which it is applied, and we do not want to specify more precisely which specific element is to be chosen. Note that each model of SET_CHOOSE will provide a function implementing some specific choice strategy, and that since all these interpretations of *choose* have to be *functions*, they are necessarily 'deterministic' (e.g., applied twice to the same set argument, they return the same result).

Datatypes with observer operations can be specified as generated instead of free.

spec SET_GENERATED [**sort** *Elem*] =
generated type $Set ::= empty \mid insert(Elem;\ Set)$
pred $\_\_is\_in\_\_ : Elem \times Set$
$\forall e, e' : Elem;\ S, S' : Set$
- $\neg(e\ is\_in\ empty)$
- $e\ is\_in\ insert(e', S) \Leftrightarrow (e = e' \vee e\ is\_in\ S)$
- $S = S' \Leftrightarrow (\forall x : Elem \bullet x\ is\_in\ S \Leftrightarrow x\ is\_in\ S')$

end

The above specification is an alternative to the specification SET given on p. 21. Both SET and SET_GENERATED define exactly the same class of models. The former specification relies on a freeness constraint, while SET_GENERATED relies on the observer *is_in* to specify when two sets are equal.

[9] For the purpose of this example, we disregard the fact that *choose* should be undefined on the empty set, and we just leave this case unspecified. Partial functions are discussed in Chap. 4.

Indeed, the last axiom of SET_GENERATED expresses directly that two sets having exactly the same elements are equal values. This axiom, together with the first two axioms defining *is_in*, will entail as well the expected properties on the constructor *insert* (see GENERATED_SET on p. 16). Note also that since, in SET_GENERATED, the predicate *is_in* is not defined within a freeness constraint, we specify when it holds using '$\Leftrightarrow$' rather than a 1-way implication.

While a freeness constraint may be unavoidable to define a predicate, as illustrated by TRANSITIVE_CLOSURE, the choice between relying on a freeness constraint to define a datatype such as *Set*, or using instead a generated datatype declaration together with some observers to unambiguously determine the values of interest, is largely a matter of convenience. One may argue that SET is more suitable for prototyping tools based on term rewriting, while SET_GENERATED is more suitable for theorem-proving tools.

The ***%def*** *annotation is useful to indicate that some operations or predicates are uniquely defined.*

spec SET_UNION [**sort** *Elem*] =
SET [**sort** *Elem*]
then %def
ops $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty$;
$remove : Elem \times Set \to Set$
$\forall e, e' : Elem;\ S, S' : Set$
- $S \cup insert(e', S') = insert(e', S \cup S')$
- $remove(e, empty) = empty$
- $remove(e, insert(e, S)) = remove(e, S)$
- $remove(e, insert(e', S)) = insert(e', remove(e, S))$ *if* $\neg(e = e')$

end

The annotation **%def** expresses that SET_UNION is a *definitional extension* of SET, i.e., that each model of SET can be *uniquely* extended to a model of SET_UNION, which means that the operations introduced in SET_UNION are uniquely defined. As the **%implies** annotation, the **%def** annotation has no impact on the semantics but merely provides useful proof obligations. The **%def** annotation is especially useful to stress that the specifier's intention is to impose a unique interpretation of what is defined within the scope of this annotation (once an interpretation for the part which is extended has been chosen).

Operations can be defined by axioms involving observer operations, instead of inductively on constructors.

spec SET_UNION_1 [**sort** *Elem*] =
SET_GENERATED [**sort** *Elem*]
then %def
ops $\_\_ \cup \_\_ : Set \times Set \to Set, assoc, comm, idem, unit\ empty$;
$remove : Elem \times Set \to Set$
$\forall e, e' : Elem;\ S, S' : Set$
• $e\ is\_in\ (S \cup S') \Leftrightarrow (e\ is\_in\ S \vee e\ is\_in\ S')$
• $e\ is\_in\ remove(e', S) \Leftrightarrow (\neg(e = e') \wedge e\ is\_in\ S)$
end

The specification SET_UNION_1 is an alternative to SET_UNION and defines exactly the same model class. While an inductive definition style was chosen for the operations '$\cup$' and *remove* in SET_UNION, in SET_UNION_1 these operations are defined 'implicitly' by characterizing their results through the observer *is_in*. Note that this 'observer' style does not prevent us providing a unique definition of both operations, as claimed by the **%def** annotation.

Similarly to the discussion on the respective merits of SET and of SET_GENERATED, the choice between an inductive definition style and an 'observer' definition style is partly a matter of taste. One may argue that the 'observer' definition style is more abstract in the sense that there is no hint to any algorithmic computation of the so-defined operations, while the inductive definition style mimics a recursive definition in a functional programming language. Again, the inductive definition style may be more suitable for prototyping tools based on term rewriting, while the 'observer' definition style may be more suitable for theorem-proving tools.

Sorts declared in free specifications are not necessarily generated by their constructors.

spec UNNATURAL =
free { **type** $UnNat ::= 0 \mid suc(UnNat)$
op $\_\_ + \_\_ : UnNat \times UnNat \to UnNat$,
$assoc, comm, unit\ 0$
$\forall x, y : UnNat \bullet x + suc(y) = suc(x + y)$
$\forall x : UnNat \bullet \exists y : UnNat \bullet x + y = 0$ **}**
end

This rather peculiar example illustrates the fact that a sort defined within a freeness constraint need not be generated by its constructors. The (unique up to isomorphism) model of UNNATURAL corresponds to integers (and not

natural numbers as one may expect) due to the last axiom. This example points out why in general datatypes defined using freeness constraints can be more difficult to understand than datatypes defined using generatedness constraints. However, the reader should be aware that the specification UnNatural uses a proper first-order formula with an existential quantifier in the axioms. The specification UnNatural is provided here for explanatory purposes only, and clearly the writing of similar specifications should be discouraged. When only Horn clauses are used as axioms in a freeness constraint, then the datatype will indeed be generated by its constructors.

4

Partial Functions

Partial functions arise naturally.

Partial functions arise in a number of systems. CASL provides means for the declaration of partial functions, the specification of their domains of definition, and more generally the specification of system properties involving partial functions. The aim of this chapter is to discuss and illustrate how to handle partial functions in CASL specifications.

4.1 Declaring Partial Functions

Partial functions are declared differently from total functions.

spec SET_PARTIAL_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **op** *choose* : *Set* →? *Elem*
end

The *choose* function on sets is naturally a partial function, expected to be undefined on the empty set. In CASL, a partial function is declared similarly to a total one, *but* for the question mark '?' following the arrow in the profile. It is therefore quite easy to distinguish the functions declared to be total from the ones declared to be partial.

A function declared to be partial may happen to be total in some of the models of the specification. For instance, the above specification SET_PARTIAL_CHOOSE does not exclude models where the function symbol *choose* is interpreted by a total function, defined on all set values. Axioms will be

used to specify the domain of definition of a partial function, and how to do this is detailed later in this chapter.

A constant may also be declared to be partial, allowing its value to be undefined in some models. Variables, however, range only over defined values.

> *Terms containing partial functions may be undefined, i.e., they may not denote any value.*

For instance, the (value of the) term $choose(empty)$ may be undefined.[1]

> *Functions, even total ones, propagate undefinedness.*

If the term $choose(S)$ is undefined for some value of S, then the term $insert(choose(S), S')$ is undefined as well for this value of S, although *insert* is a total function.

> *Predicates do not hold on undefined arguments.*

CASL is based on classical two-valued logic. A predicate symbol is interpreted by a relation, and when the value of some argument term is undefined, the application of a predicate to this term does not hold. For instance, if the term $choose(S)$ is undefined, then the atomic formula $choose(S)$ *is_in* S does not hold.

> *Equations hold when both terms are undefined.*

In CASL, equations are by default *strong*, which means that they hold not only when both sides denote equal values, but also when both sides are simultaneously undefined. For instance, let us consider the equation:

$$insert(choose(S), insert(choose(S), empty)) = insert(choose(S), empty)\,.$$

Either $choose(S)$ is defined and then both sides are defined and denote equal values due to the axioms on *insert*, or $choose(S)$ is undefined and then both sides are undefined, and the strong equation 'holds trivially'.

[1] Note that the term $choose(empty)$ is well-formed and therefore is a 'correct term'. It is *its value* which may be undefined. To avoid unnecessary pedantry, in the following we will simply write that a term is undefined to mean that its value is so. Obviously, a term with variables may be defined for some values of the variables and undefined for other values.

CASL provides also so-called *existential equations*, explained at the end of this chapter.

Special care is needed in specifications involving partial functions.

Partial functions are intrinsically more difficult to understand and specify than total ones. This is why special care is needed when writing the axioms of specifications involving partial functions. The point is that an axiom may imply the definedness of terms containing partial functions, and as a consequence that these functions are total, which may not be what the specifier intended. There are three typical cases of such situations:

- Asserting $choose(S)$ is_in S as an axiom implies that $choose(S)$ is defined, for any S. The point here is that since predicates applied to an undefined term do not hold, in any model satisfying $choose(S)$ is_in S, the function *choose* must be total (i.e., always defined).
- Asserting $remove(choose(S), insert(choose(S), empty)) = empty$ as an axiom implies that $choose(S)$ is defined for any S, since the term *empty* is always defined. To understand this, assume that *choose* is undefined for some set value of S; then the above equation cannot hold for this value, since the undefinedness of $choose(S)$ implies the undefinedness of $remove(choose(S), insert(choose(S), empty))$, giving a contradiction with the definedness of *empty*. Hence, an equation between a term involving a partial function PF and a term involving total functions only may imply that the partial function PF is always defined.
- Asserting $insert(choose(S), S) = S$ as an axiom implies that $choose(S)$ is defined for any S, since a variable always denotes a defined value. This case is indeed similar to the previous one, the only difference being that now the right-hand side of the equation is a variable (instead of a term involving total functions only).

Moreover, the 'same name, same thing' principle has a subtle side-effect regarding partial operations: if an operation is declared both as a total operation and as a partial operation with the same profile (i.e. the same argument sorts and the same result sort) then it is interpreted as a total operation in all models of the specification.

4.2 Specifying Domains of Definition

The definedness of a term can be checked or asserted.

spec SET_PARTIAL_CHOOSE_1 [**sort** *Elem*] =
SET_PARTIAL_CHOOSE [**sort** *Elem*]
then • $\neg(def\ choose(empty))$
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

A definedness assertion, written '*def t*', where t is a term, is a special kind of atomic formula: it holds if and only if the value of the term t is defined. For instance, in the above example, $\neg(def\ choose(empty))$ explicitly asserts that *choose* is undefined when applied to *empty*. Note that this axiom does not say anything about the definedness of *choose* applied to values other than *empty*, which means that *choose* may well be undefined on those values too. The second axiom of the above example asserts $choose(S)\ is\_in\ S$ under the condition $def\ choose(S)$, to avoid undesired definedness induced by axioms, as explained above.

Note that if the two axioms of the above example were to be replaced by $\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$, then we could conclude that $choose(S)$ is defined when S is not equal to *empty*, but nothing about the undefinedness of $choose(empty)$.

The domains of definition of partial functions can be specified exactly.

spec SET_PARTIAL_CHOOSE_2 [**sort** *Elem*] =
SET_PARTIAL_CHOOSE [**sort** *Elem*]
then $\forall S : Set \bullet def\ choose(S) \Leftrightarrow \neg(S = empty)$
$\forall S : Set \bullet def\ choose(S) \Rightarrow choose(S)\ is\_in\ S$
end

In the above example, the domain of definition of the partial function *choose* is exactly specified by the axiom $def\ choose(S) \Leftrightarrow \neg(S = empty)$.

Loosely specified domains of definition may be useful.

spec NATURAL_WITH_BOUND_AND_ADDITION =
NATURAL_WITH_BOUND
then **op** $\_\_+?\_\_ : Nat \times Nat \rightarrow ?\ Nat$
$\forall x, y : Nat$
• $def(x+?y)\ if\ x + y < max\_size$
%{ $x + y < max\_size$ implies both
$x < max\_size$ and $y < max\_size$ **}%**
• $def(x+?y) \Rightarrow x+?y = x + y$
end

In some cases, it is useful to loosely specify the domain of definition of a partial function, as illustrated in the above example for '+?', which is required to be defined for all arguments x and y such that $x + y < max\_size$, but may well be defined on larger natural numbers as well. The point in loose specifications of definition domains is to avoid unnecessary constraints on the models of the specification. For instance, the above example does not exclude a model where '+?' is interpreted by a total function (which would then coincide with '+').[2]

Indeed, in some cases, specifying exactly domains of definition can be considered as overspecification. In most specifications, however, one would expect an exact specification of domains of definition, even for otherwise loosely specified functions (see, e.g., *choose* in SET_PARTIAL_CHOOSE_2).

Domains of definition can be specified more or less explicitly.

spec SET_PARTIAL_CHOOSE_3 [**sort** *Elem*] =
SET_PARTIAL_CHOOSE [**sort** *Elem*]
then • $\neg(def\ choose(empty))$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S)\ is\_in\ S$
end

SET_PARTIAL_CHOOSE_3 specifies exactly the domain of definition of *choose*, but does this too implicitly, since some reasoning is needed to conclude that the above specification entails $def\ choose(S) \Leftrightarrow \neg(S = empty)$. To improve the clarity of specifications, it is in general advisable to specify definition domains as explicitly as possible, and SET_PARTIAL_CHOOSE_2 is somehow easier to understand than SET_PARTIAL_CHOOSE_3 (both specifications define the same class of models).

spec NATURAL_PARTIAL_PRE =
NATURAL_ARITHMETIC
then **op** $pre : Nat \to? Nat$
$\forall x, y : Nat$
• $\neg\ def\ pre(0)$
• $pre(suc(x)) = x$
end

In the above example, one can consider that the domain of definition of *pre* is (exactly) specified in an explicit enough way, since the first axiom states exactly that $pre(0)$ is undefined while the second one implies that *pre* is defined for all natural numbers of the form $suc(x)$.

[2] In this example, it is essential to choose a new name '+?' for our partial addition operation. Otherwise, since '+' is (rightly) declared as a total operation in NATURAL_WITH_BOUND, the declaration **op** $\_\_ + \_\_ : Nat \times Nat \to? Nat$ would be useless: the same name, same thing principle would lead to models with just one, total, addition operation.

spec NATURAL_PARTIAL_SUBTRACTION =
NATURAL_PARTIAL_PRE
then **op** $\_\_ - \_\_ : Nat \times Nat \to ? Nat$
$\forall x, y : Nat$
- $x - 0 = x$
- $x - suc(y) = pre(x - y)$

end

The above specification is perfect from a mathematical point of view, but is clearly not explicit enough, since there is no easy way to infer when $x - y$ is defined. From a methodological point of view, the following alternative version is much better.

spec NATURAL_PARTIAL_SUBTRACTION_1 =
NATURAL_PARTIAL_PRE
then **op** $\_\_ - \_\_ : Nat \times Nat \to ? Nat$
$\forall x, y : Nat$
- $def(x - y) \Leftrightarrow (y < x \vee y = x)$
- $x - 0 = x$
- $x - suc(y) = pre(x - y)$

end

The above examples clearly demonstrate why the explicit specification of definition domains is generally advisable from a methodological point of view. However, they also indicate that this recommendation should not be applied in too strict a way, and that deciding whether a specification is explicit enough or not is to some extent a matter of taste.

Partial functions are minimally defined by default in free specifications.

spec LIST_SELECTORS_1 [**sort** *Elem*] =
LIST [**sort** *Elem*]
then **free** { **ops** $head : List \to ? Elem$;
$tail : List \to ? List$
$\forall e : Elem;\ L : List$
- $head(cons(e, L)) = e$
- $tail(cons(e, L)) = L$ }

end

In the above example, the given axioms imply that *head* and *tail* are defined on lists of the form $cons(e, L)$. The freeness constraint implies that these functions are minimally defined. Since the terms $head(empty)$ and $tail(empty)$ are not equated to any other term, the freeness constraint implies that these terms are undefined, and hence that the functions *head* and *tail* are undefined on *empty*. The situation here is similar to the fact that predicates hold minimally in models of free specifications (see Chap. 3, p. 19).

spec LIST_SELECTORS_2 [**sort** *Elem*] =
LIST [**sort** *Elem*]
then **ops** $head : List \to ? Elem$;
$tail : List \to ? List$
$\forall e : Elem;\ L : List$
- $\neg\, def\ head(empty)$
- $\neg\, def\ tail(empty)$
- $head(cons(e, L)) = e$
- $tail(cons(e, L)) = L$

end

The above specification LIST_SELECTORS_2 is an alternative to LIST_SELECTORS_1, both specifications define exactly the same class of models. However, LIST_SELECTORS_2 is clearly easier to understand and can be considered as technically simpler, since it involves no freeness constraint.

Operations like *head* and *tail* are usually called *selectors*, and CASL provides abbreviations to specify selectors in a very concise way, as we see next.

4.3 Partial Selectors and Constructors

Selectors can be specified concisely in datatype declarations, and are usually partial.

spec LIST_SELECTORS [**sort** *Elem*] =
free type $List ::= empty \mid cons(head\ :?\ Elem;\ tail\ :?\ List)$
end

The above free datatype declaration introduces, in addition to the constructors *empty* and *cons*, two partial selectors *head* and *tail* yielding the respective arguments of the constructor *cons*. Hence, this free datatype declaration with selectors has exactly the same effect as the ordinary free datatype declaration **free type** $List ::= empty \mid cons(Elem;\ List)$, together with the operation declarations and axioms of LIST_SELECTORS_2 (i.e., LIST_SELECTORS and LIST_SELECTORS_2 define exactly the same class of models). The following example is similar in spirit.

spec NATURAL_SUC_PRE =
free type $Nat ::= 0 \mid suc(pre\ :?\ Nat)$
end

Selectors are usually total when there is only one constructor.

spec PAIR [**sort** *Elem1*] [**sort** *Elem2*] =
free type $Pair ::= pair(first : Elem1;\ second : Elem2)$
end

While selectors are usually partial operations when there is more than one alternative in the corresponding datatype declaration, they can be total, and this is generally the case when there is only one constructor, as in the above example. The free datatype declaration entails in particular axioms asserting that *first* and *second* yield the respective arguments of the constructor *pair* (i.e., $first(pair(e1, e2)) = e1$ and $second(pair(e1, e2)) = e2$).

Constructors may be partial.

spec PART_CONTAINER [**sort** *Elem*] =
generated type
$P\_Container ::= empty \mid insert(Elem;\ P\_Container)?$
pred $addable : Elem \times P\_Container$
vars $e, e' : Elem;\ C : P\_Container$
• $def\ insert(e, C) \Leftrightarrow addable(e, C)$
pred $\_\_is\_in\_\_ : Elem \times P\_Container$
• $\neg(e\ is\_in\ empty)$
• $(e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C))\ if\ addable(e', C)$
end

The intention in the above example is to define a reusable specification of partial containers. The *insert* constructor is specified as a partial operation, defined if some condition on both the element e to be added and the container C to which the element is to be added holds. This condition is abstracted here in a predicate *addable*, so far left unspecified. Later on, instantiations of the PART_CONTAINER specification can be adapted to specific purposes by extending them with axioms defining *addable*.

The above generated datatype declaration abbreviates as usual the declaration of a sort $P\_Container$, a constant constructor *empty*, and a partial constructor $insert : Elem \times P\_Container \to? P\_Container$. It also entails the corresponding generatedness constraint.

4.4 Existential equality

Existential equality involves the definedness of both terms as well as their equality.

spec NATURAL_PARTIAL_SUBTRACTION_2 =
NATURAL_PARTIAL_SUBTRACTION_1
then $\forall x, y : Nat \bullet y - x \stackrel{e}{=} z - x \Rightarrow y = z$
%{ $y - x = z - x \Rightarrow y = z$ would be wrong,
$def(y - x) \wedge def(z - x) \wedge y - x = z - x \Rightarrow y = z$
is correct, but better abbreviated in the above axiom }%
end

An *existential* equation $t1 \stackrel{e}{=} t2$ is equivalent to $def(t1) \wedge def(t2) \wedge t1 = t2$, so it holds if and only if both terms *t1* and *t2* are defined and denote the same value.

Note that while a trivial strong equation of the form $t = t$ always holds, this is not the case for existential equations. For instance, the trivial existential equation $x - y \stackrel{e}{=} x - y$ does not hold, since the term $x - y$ may be undefined.

In general consequences of undefinedness are undesirable. Hence a conditional equation of the form $t1 = t2 \Rightarrow t3 = t4$ is often wrong if *t1* and *t2* may be undefined, because the equality $t3 = t4$ would be implied when both *t1* and *t2* are undefined (since then the strong equation $t1 = t2$ would hold). The above specification provides a typical example of such a situation: $y - x = z - x \Rightarrow y = z$ would be wrong, since it would entail that any two arbitrary values y and z are equal (it is enough to choose an x greater than y and z to make $y - x$ and $z - x$ both undefined).

Therefore, to avoid such undesirable consequences of undefinedness, it is advisable to use existential equations instead of strong equations in the premises of conditional equations involving partial operations. An alternative is to add the relevant definedness assertions explicitly to the equations in the premises.

5

Subsorting

Subsorts and supersorts are often useful in CASL *specifications.*

Many examples naturally involve subsorts and supersorts. CASL provides means for the declaration of a sort as a subsort of another one when the values of the subsort are regarded a special case of those in the other sort. The aim of this chapter is to discuss and illustrate how to handle subsorts and supersorts in CASL specifications.

5.1 Subsort Declarations and Definitions

Subsort declarations directly express relationships between carrier sets.

spec GENERIC_MONOID_1 [**sort** *Elem*] =
sorts *Elem* < *Monoid*
ops *1* : *Monoid*;
_ _ * _ _ : *Monoid* × *Monoid* → *Monoid*, *assoc*, *unit 1*
end

The above example declares the sort *Elem* to be a subsort of *Monoid*, or, symmetrically, the sort *Monoid* to be a supersort of *Elem*. Hence the specification GENERIC_MONOID_1 is quite similar to the specification GENERIC_MONOID given in Chap. 3, p. 11, the only difference being the use of a subsorting relation between *Elem* and *Monoid* instead of an explicit *inj* operation to embed values of sort *Elem* into values of sort *Monoid*.

In contrast to most other algebraic specification languages providing subsorting facilities, subsorts in CASL are interpreted by arbitrary *embeddings*

between the corresponding carrier sets. In the above example, the subsort declaration $Elem < Monoid$ induces an implicit (unnamed) embedding from the carrier of the sort *Elem* into the carrier of the sort *Monoid*. Thus the main difference between GENERIC_MONOID and GENERIC_MONOID_1 is that the embedding is explicit and named *inj* in GENERIC_MONOID while it is implicit in GENERIC_MONOID_1.

Note that interpreting subsorting relations by embeddings rather than inclusions does not exclude models where the (carrier of the) subsort happens to be a subset of (the carrier of) the supersort, and the embedding a proper inclusion. Embeddings are just slightly more general than inclusions, and technically not much more complex.

Operations declared on some sort are automatically inherited by its subsorts.

spec VEHICLE =
NATURAL
then **sorts** $Car, Bicycle < Vehicle$
ops $max\_speed : Vehicle \to Nat$;
$weight : Vehicle \to Nat$;
$engine\_capacity : Car \to Nat$
end

The above example introduces three sorts, *Car*, *Bicycle* and *Vehicle*, and declares both *Car* and *Bicycle* to be subsorts of *Vehicle*. A subsort declaration entails that any term of a subsort is also a term of the supersort, so here, any term of sort *Car* is also a term of sort *Vehicle*, and we can apply the operations *max_speed* and *weight* to it (and similarly for a term of sort *Bicycle*).

In other words, with the single declaration $max\_speed : Vehicle \to Nat$, we get the effect of having declared also two other operations, $max\_speed : Car \to Nat$ and $max\_speed : Bicycle \to Nat$.[1]

Obviously, operations that are only meaningful for some subsort should be defined at the appropriate level. This is the case here for the operation *engine_capacity*, which is only relevant for cars, and therefore defined with the appropriate profile exploiting the subsort *Car*.

Inheritance applies also for subsorts that are declared afterwards.

spec MORE_VEHICLE = VEHICLE **then sorts** $Boat < Vehicle$ **end**

[1] Strictly speaking, there is just one *max_speed* operation in the signature of VEHICLE.

The order in which a subsort and an operation on the supersort are declared is irrelevant. In MORE_VEHICLE, we introduce a further subsort *Boat* of *Vehicle*, and as a consequence, we again get the effect of having both *max_speed* and *weight* available for boats, as it was already the case for cars and bikes.

Subsort membership can be checked or asserted.

spec SPEED_REGULATION =
VEHICLE
then **ops** $speed\_limit : Vehicle \to Nat$;
$car\_speed\_limit, bike\_speed\_limit : Nat$
$\forall v : Vehicle$
- $v \in Car \Rightarrow speed\_limit(v) = car\_speed\_limit$
- $v \in Bicycle \Rightarrow speed\_limit(v) = bike\_speed\_limit$

end

A subsort membership assertion, written '$t \in s$', where t is a term and s is a sort, is a special kind of atomic formula: it holds if and only if the value of the term t is the embedding of some value of sort s. For instance, in the above example, $v \in Car$ holds if and only if v denotes a vehicle which is the embedding of a car value. Note that '$\in$' is input as '`in`', but displayed as '$\in$'.

Datatype declarations can involve subsort declarations.

The specification:

sorts *Car*, *Bicycle*, *Boat*
type *Vehicle* ::= *sort Car* | *sort Bicycle* | *sort Boat*

is equivalent to the declaration **sorts** $Car, Bicycle, Boat < Vehicle$. There may be some values of sort *Vehicle* which are not the embedding of any value of sort *Car*, *Bicycle*, or *Boat*. Intuitively, the above datatype declaration just means that *Vehicle* 'contains' the *union* (which may not be disjoint) of *Car*, *Bicycle* and *Boat*. Note that the subsorts used in the datatype declaration must already be declared beforehand.

The specification:

sorts *Car*, *Bicycle*, *Boat*
generated type *Vehicle* ::= *sort Car* | *sort Bicycle* | *sort Boat*

is more restrictive, since the generatedness constraint implies that any value of the supersort *Vehicle* must be the embedding of some value of the declared subsorts *Car*, *Bicycle* and *Boat*. Intuitively, the above datatype declaration means that *Vehicle* 'is exactly' the union (which again may not be disjoint) of *Car*, *Bicycle* and *Boat*. In particular, this declaration prevents subsequent

introduction of further subsorts (unless the values of the new subsorts are intended to correspond to some values of the already declared subsorts). For instance, if we were now to extend the above specification with **sorts** $Plane < Vehicle$, all values of sort *Plane* would have to correspond to *Car*, *Bicycle* or *Boat* values (which is presumably not what we were intending).

The specification:

sorts *Car*, *Bicycle*, *Boat*
free type $Vehicle ::= sort\ Car \mid sort\ Bicycle \mid sort\ Boat$

entails the same generatedness constraint as in the previous example, and, moreover, the freeness constraint requires that there is no 'common' value in the subsorts of *Vehicle*. Intuitively, the above declaration means that *Vehicle* 'is exactly' the *disjoint* union of *Car*, *Bicycle* and *Boat*. This means in particular that the introduction of a further common subsort of both *Car* and *Boat* (say, **sorts** $Amphibious < Car, Boat$) is impossible.

Subsorts may also arise as classifications of previously specified values, and their values can be explicitly defined.

spec NATURAL_SUBSORTS =
NATURAL_ARITHMETIC
then **pred** $even : Nat$
• $even(0)$
• $\neg\, even(1)$
$\forall n : Nat \bullet even(suc(suc(n))) \Leftrightarrow even(n)$
sort $Even = \{x : Nat \bullet even(x)\}$
sort $Prime = \{x : Nat \bullet \forall y, z : Nat \bullet x = y * z \Rightarrow y = 1 \vee z = 1\}$
end

The subsort definition **sort** $Even = \{x : Nat \bullet even(x)\}$ is equivalent to the declaration of a subsort *Even* of *Nat* (i.e., **sorts** $Even < Nat$) together with the assertion $\forall x : Nat \bullet x \in Even \Leftrightarrow even(x)$.

The main advantage of defining the subsort *Even* in addition to the predicate *even* is that we may then use the subsort when declaring operations (e.g., **op** $times2 : Nat \to Even$) and variables.

The subsort definition for *Prime* above illustrates that it is not always necessary to introduce and define an explicit predicate characterizing the values of the subsort: the formula used in a subsort definition is not restricted to predicate applications. In fact whenever a (unary) predicate p on a sort s could be defined by **pred** $p(x : s) \Leftrightarrow f$ for some formula f, we may instead define **sort** $P = \{x : s \bullet f\}$, and use sort membership assertions $t \in P$ instead of predicate applications $p(t)$, avoiding the introduction of the predicate p altogether.

The following example is a further illustration of subsort definitions. We declare a subsort *Pos* of *Nat* and ensure that values of *Pos* correspond to

non-zero values of *Nat*. (Several alternative ways of specifying the sort *Pos* will be considered later in this section.)

spec POSITIVE =
NATURAL_PARTIAL_PRE
then **sort** $Pos = \{x : Nat \bullet \neg(x = 0)\}$ **end**

5.2 Subsorts and Overloading

It may be useful to redeclare previously defined operations, using the new subsorts introduced.

spec POSITIVE_ARITHMETIC =
POSITIVE
then **ops** $1 : Pos$;
$suc : Nat \to Pos$;
$\_\_ + \_\_, \_\_ * \_\_ : Pos \times Pos \to Pos$;
$\_\_ + \_\_ : Pos \times Nat \to Pos$;
$\_\_ + \_\_ : Nat \times Pos \to Pos$
end

Since, in POSITIVE, we have defined *Pos* as a subsort of *Nat*, all operations defined on natural numbers, like *suc*, '+' and '∗' (see NATURAL_ARITHMETIC in Chap. 3, p. 20, which is extended into NATURAL_PARTIAL_PRE in Chap. 4, p. 31), are automatically inherited by *Pos* and can be applied to terms of sort *Pos*. However, according to their declarations, these operations, when applied to terms of sort *Pos*, yield results of sort *Nat*. To indicate that these results always correspond to values in the subsort *Pos*, it is necessary to explicitly *overload* these operations by similar ones with the appropriate profiles. This is the aim of the first three lines of operation declarations in the above example. The last two operation declarations further overload '+' to specify that '+' also yields a result of sort *Pos* as soon as one of its arguments is a term of sort *Pos*.

It is quite important to understand that the above overloading declarations are enough to achieve the desired effect, and that *no axioms are necessary*. The fundamental rule is that, in models of CASL specifications with subsorting, embedding and overloading have to be compatible: embeddings commute with overloaded operations. This can be rephrased into the following intuitive statement: *If terms look the same, then they have the same value in the same sort*. Thus, in our example, the value of '*1 + 1*' is the same in *Nat* whatever the combination of the overloaded constant '*1*' and operation '+' is chosen, and there is no need for any axiom to ensure this, since this is implicit in the semantics of subsorting.

5.3 Subsorts and Partiality

A subsort may correspond to the definition domain of a partial function.

spec POSITIVE_PRE =
POSITIVE_ARITHMETIC
then **op** $pre : Pos \to Nat$ **end**

Since we have introduced the subsort *Pos* of non-zero natural numbers, it makes sense to overload the partial *pre* operation on *Nat* by a *total* one on *Pos*, as illustrated above, to emphasize the fact that indeed *pre* is a total operation on its definition domain. Note again that no further axiom is necessary, and that the semantics of subsorting will ensure that both the partial and total *pre* operations will give the same value when applied to the same non-zero value.[2]

Using subsorts may avoid the need for partial functions.

spec NATURAL_POSITIVE_ARITHMETIC =
free types $Nat ::= 0 \mid sort\ Pos$;
$Pos ::= suc(pre : Nat)$
ops $1 : Pos = suc(0)$;
$\_\_ + \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 0$;
$\_\_ * \_\_ : Nat \times Nat \to Nat, assoc, comm, unit\ 1$;
$\_\_ + \_\_, \_\_ * \_\_ : Pos \times Pos \to Pos$;
$\_\_ + \_\_ : Pos \times Nat \to Pos$;
$\_\_ + \_\_ : Nat \times Pos \to Pos$
$\forall x, y : Nat$
- $x + suc(y) = suc(x + y)$
- $x * 0 = 0$
- $x * suc(y) = x + (x * y)$

end

It is indeed tempting to exploit subsorting to avoid the declaration of partial functions, as illustrated by the above NATURAL_POSITIVE_ARITHMETIC specification, which is an alternative to POSITIVE_PRE and avoids the introduction of the partial predecessor operation. Note that in the above example, we have fully used the facilities for defining free datatypes with subsorts, and

[2] This should not be confused with the same name, same meaning principle, which does not apply here: the total *pre* and the partial one have different profiles, and hence are just overloaded.

in particular non-linear visibility for the declared sorts, which allows us to refer to the subsort *Pos* in the first line before defining it in the second one.

Avoiding the introduction of the partial predecessor operation has some drawbacks, since some previously well-formed terms (with defined values) have now become ill-formed, e.g., $pre(pre(suc(1)))$, where $pre(suc(1))$ is a (well-formed) term of sort *Nat* but *pre* expects an argument of sort *Pos*. (The fact that $pre(suc(1)) = 1$ is a consequence of the specified axioms and that *1* is of sort *Pos* does not of course entail that $pre(suc(1))$ is of sort *Pos* too, since axioms are disregarded when checking for well-formedness.) See below for possible workarounds using explicit casts. Moreover, it is not always possible, or easy, to avoid the declaration of partial operations by using appropriate subsorts—just consider, e.g., subtraction on natural numbers.

As a last remark on this issue, the reader should be aware of the fact that, while overloading a partial operation on a supersort (say, *pre* on *Nat*) with a total one on a subsort (*pre* on *Pos*) is fine, overloading a total operation on a *supersort* with a partial one on a *subsort* forces the partial operation to be total, and hence, the latter would be better declared as total too.[3]

Casting a term from a supersort to a subsort is explicit and the value of the cast may be undefined.

In CASL, a term of a subsort can always be considered as a term of any supersort, and embeddings are implicit. On the contrary, casting a term from a supersort to a subsort is explicit, and since casting is essentially a partial operation, the resulting casted term may not denote any value. Casting a term t to a sort s is written $t\ as\ s$. Consider the term $pre(\ pre(suc(1))\ as\ Pos\ )$ which is well-formed in the context of the NATURAL_POSITIVE_ARITHMETIC specification. This term does indeed denote a value, but the value is not a positive natural number, so the value of the term $pre(\ pre(suc(1))\ as\ Pos\ )\ as\ Pos$ is undefined.

Note that $def\ (t\ as\ s)$ is equivalent to $t \in s$, for a well-formed term t of a supersort of s.

[3] Overloading a total *cons* on *List* with a partial *cons* on the subsort *OrderedList* would either lead to a total *cons* operation on *OrderedList*, or to an inconsistent specification, depending on how the definition domain of the partial *cons* is specified.

Supersorts may be useful when generalizing previously specified sorts.

spec INTEGER_ARITHMETIC_1 =
NATURAL_POSITIVE_ARITHMETIC
then **free type** $Int ::= sort\ Nat \mid -\_\_(Pos)$
ops $\_\_ + \_\_ : Int \times Int \to Int, assoc, comm, unit\ 0;$
$\_\_ - \_\_ : Int \times Int \to Int;$
$\_\_ * \_\_ : Int \times Int \to Int, assoc, comm, unit\ 1;$
$abs : Int \to Nat$

$\forall x, y : Int;\ n : Nat;\ p, q : Pos$
- $suc(n) + (-1) = n$
- $suc(n) + (-suc(q)) = n + (-q)$
- $(-p) + (-q) = -(p + q)$
- $x - 0 = x$
- $x - p = x + (-p)$
- $x - (-q) = x + q$
- $n * (-q) = -(n * q)$
- $(-p) * (-q) = p * q$
- $abs(n) = n$
- $abs(-p) = p$

end

The specification INTEGER_ARITHMETIC_1 extends NATURAL_POSITIVE_ARITHMETIC and defines the sort *Int* as a supersort of the sort *Nat*. As a consequence, terms which have two parses in sort *Int*, such as $suc(0)$, are required by the semantics of subsorting to have the same value for both parses, and they can therefore be used without explicit disambiguation.

The situation would be quite different if one would be using a combination of NATURAL_ARITHMETIC and INTEGER_ARITHMETIC (see Chap. 3), say by extending both in a structured specification (see the next chapter for more details on structured specifications). In such a combination, a term such as $suc(0)$ would have two parses, one in sort *Nat* and one in sort *Int*; in the absence of any subsort declaration relating *Nat* and *Int*, and hence of any implicit embedding, this term would then be ambiguous, and would require explicit disambiguation to become a well-formed term.

Supersorts may also be used for extending the intended values by new values representing errors or exceptions.

spec SET_ERROR_CHOOSE [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
pred $\_\_is\_in\_\_ : ElemError \times Set$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow choose(S) \in Elem \wedge choose(S)\ is\_in\ S$
end

The above specification SET_ERROR_CHOOSE is another variant of the various specifications of sets equipped with a partial choose function given in Chap. 4. This variant avoids the declaration of a partial function *choose* by using instead a supersort of *Elem*, namely *ElemError*, as the target sort of *choose*. The idea here is that values of *ElemError* which are not (embeddings of) values of *Elem* represent errors, e.g., the application of *choose* to the empty set. Note that to obtain the desired effect, it is necessary to explicitly state that $choose(S) \in Elem$ when S is not the empty set; moreover, to make the term $choose(S)\ is\_in\ S$ well-formed, we have to explicitly overload the predicate $\_\_is\_in\_\_ : Elem \times Set$ provided by GENERATED_SET by a predicate $\_\_is\_in\_\_ : ElemError \times Set$ as shown above. This example demonstrates that avoiding partial functions by the use of 'error supersorts' is not as innocuous as it may seem, since in general one would need to enlarge the signatures considerably by adding all required overloadings.

spec SET_ERROR_CHOOSE_1 [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*]
then **sorts** $Elem < ElemError$
op $choose : Set \to ElemError$
$\forall S : Set \bullet \neg(S = empty) \Rightarrow (choose(S)\ as\ Elem)\ is\_in\ S$
end

The specification SET_ERROR_CHOOSE_1 above is a last attempt to avoid the use of partial functions; again, we introduce a supersort *ElemError* as in SET_ERROR_CHOOSE, but to avoid the need for overloading the predicate *is_in*, we explicitly *cast* the term $choose(S)$ in $(choose(S)\ as\ Elem)\ is\_in\ S$. Note that when, for some value of S, $(choose(S)\ as\ Elem)\ is\_in\ S$ holds, this implies that $choose(S)\ as\ Elem$ is defined, and hence that $choose(S) \in Elem$ holds as well. This last version may seem preferable to the previous one. However, one should be aware that here, despite our attempt to avoid the use of partial functions, we rely on explicit casts, hence on terms that may not denote values: partiality has not been eliminated, the partial functions have merely been factorized as compositions of total functions with casting.

6

Structuring Specifications

Large and complex specifications are easily built out of simpler ones by means of (a small number of) specification-building operations.

In the previous chapters, we have focussed attention on basic specifications and detailed how to use the various constructs of CASL to write meaningful, but relatively simple, specifications. The aim of this chapter is to discuss and illustrate how to assemble simple pieces of specifications into more complex, structured ones. In particular we explain how to extend specifications, make the union of several specifications, as well as how to rename or hide symbols when assembling specifications. Parametrization and instantiation of generic specifications are explained in the next chapter.

6.1 Union and Extension

Union and extension can be used to structure specifications.

spec LIST_SET [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **and**
GENERATED_SET [**sort** *Elem*]
then **op** $elements\_of\_\_ : List \to Set$
$\forall e : Elem;\ L : List$
• $elements\_of\ empty = empty$
• $elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$
end

The above example shows how to make the union (expressed by '**and**') of two specifications LIST_SELECTORS (see Chap. 4, p. 33) and GENERATED_SET (see Chap. 3, p. 16), and then further extend this union by some operation and axioms (using '**then**'). Union and extension are the most used specification-building operations. In contrast with extension, whose purpose is to extend a given piece of specification by new symbols and axioms, union is generally used to combine two self-contained specifications. Union of specifications is obviously an associative and commutative operation.

All symbols introduced by a specification are by default exported by it and visible in its extensions or in its unions with other specifications. (Variables are not considered as symbols, and never exported.) Remember also the '*same name, same thing*' principle: in the above LIST_SET specification, it is therefore the same sort *Elem* which is used to construct both lists and sets.

Arbitrary parts of specifications can have initial semantics.

spec LIST_CHOOSE [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **and**
SET_PARTIAL_CHOOSE_2 [**sort** *Elem*]
then **ops** $elements\_of\_\_ : List \to Set$;
$choose : List \to ? Elem$
$\forall e : Elem;\ L : List$
- $elements\_of\ empty = empty$
- $elements\_of\ cons(e, L) = \{e\} \cup elements\_of\ L$
- $def\ choose(L) \Leftrightarrow \neg(L = empty)$
- $choose(L) = choose(elements\_of\ L)$

end

spec SET_TO_LIST [**sort** *Elem*] =
LIST_SET [**sort** *Elem*]
then **op** $list\_of\_\_ : Set \to List$
$\forall S : Set \bullet elements\_of(list\_of\ S) = S$
end

The specification LIST_CHOOSE is built as an extension of the union of LIST_SELECTORS and SET_PARTIAL_CHOOSE_2 (see Chap. 4, p. 30). This extension introduces an operation *elements_of* (as in LIST_SET) and a partial operation *choose*, which are defined by some axioms. In LIST_SELECTORS, lists are defined by a free datatype construct (with selectors), hence have a free interpretation. SET_PARTIAL_CHOOSE_2 is itself an extension of (SET_PARTIAL_CHOOSE which is an extension of) GENERATED_SET, where sets are defined by a generated datatype construct. However, note that as discussed in Chap. 3, p. 16, the apparently loose specification GENERATED_SET is in fact not so. Moreover, the *choose* partial function on sets is loosely defined in SET_PARTIAL_CHOOSE_2, and so is therefore also the *choose* partial function on

lists defined in LIST_CHOOSE. It is easy to see that the operation *elements_of* is uniquely defined. The sort *Elem* has of course a loose interpretation.

Thus the specification LIST_CHOOSE combines parts with a free interpretation, parts with a generated interpretation, and parts with a loose interpretation. The situation is similar to that with LIST_SET (and SET_TO_LIST), where the operation *list_of* is loosely defined with the help of the operation *elements_of*.

6.2 Renaming

Renaming may be used to avoid unintended name clashes, or to adjust names of sorts and change notations for operations and predicates.

spec STACK [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **with sort** *List* ↦ *Stack*,
ops *cons* ↦ *push__onto__*,
head ↦ *top*,
tail ↦ *pop*
end

While the 'same name, same thing' principle has proven to be appropriate in numerous examples given in the previous chapters and above, it may still happen that, when combining specifications, this principle leads to unintended name clashes. An unintended name clash arises for instance when one combines two specifications that both export the same symbol (with the same profile in case of an operation or a predicate), while this symbol is not intended to denote the same 'thing' in the combination. In such cases, it is necessary to explicitly *rename* some of the symbols exported by the specifications put together in order to avoid the unintended name clashes.

When reusing a named specification, it may be convenient to rename some of its symbols; moreover, in the case of operation or predicate symbols, one may also change the style of notation. This is illustrated in the specification STACK above which is obtained as a renaming of the specification LIST_SELECTORS. First, the sort *List* is renamed into *Stack*, then the operation *cons* is renamed into a mixfix operation *push__onto__*, and finally the selectors *head* and *tail* are renamed into *top* and *pop*, respectively. Note that '↦' is input as '`|->`'.

The user only needs to indicate how symbols provided by the renamed specification are mapped to new symbols. A *signature morphism* is automatically deduced from this 'symbol map'. For instance, the signature morphism inferred from the symbol map specified in STACK maps the operation symbol *cons* : *Elem* × *List* → *List* to the operation symbol *push__onto__* :

Elem × *Stack* → *Stack*: not only the operation name is changed, but also its profile according to the renaming of *List* into *Stack*.

In a symbol map, one can qualify the symbol to be renamed by its kind, using the keywords **sort**, **op**, and **pred** (or their plural forms), as appropriate; this is illustrated in STACK above. Qualification in symbol maps is generally recommended since it improves their readability.

While it is possible to change the syntax of an operation or predicate symbol, as illustrated above for *cons* mapped to *push__onto__*, it is not possible to change the order of the arguments of the renamed operation or predicate.

In general, one does not need to rename all the symbols provided by the specification to be renamed. In the symbol map describing the intended renaming, it is indeed enough to mention only the symbols that change. By default, any symbol not explicitly mentioned is left unchanged (although its profile may be updated according to the renaming specified for some sorts). This is illustrated here in STACK where there is no need to rename the constant *empty*, which will therefore have the same name for both lists and stacks. However, the induced signature morphism maps the constant symbol *empty* : *List* into the constant symbol *empty* : *Stack*.

One can also explicitly rename a symbol to itself, say by writing '*empty* ↦ *empty*', or just mention it without providing a new name, as in '**with** *empty*', which is equivalent to '**with** *empty* ↦ *empty*'.

By default, overloaded symbols are renamed simultaneously. For instance, in INTEGER_ARITHMETIC_1 **with** __ + __ ↦ *plus*, all the five overloaded infix '+' operations exported by INTEGER_ARITHMETIC_1 (see Chap. 5, p. 44) are renamed into five *plus* operations, with a functional syntax and the appropriate profiles.

It is possible as well to reduce the overloading, or to specifically rename one of the overloaded symbols, by specifying its profile in the symbol map. For instance, in INTEGER_ARITHMETIC_1 **with** __ + __ : *Pos* × *Pos* → *Pos* ↦ *plus*, only the addition of two positive natural numbers is renamed into *plus*.

When combining specifications, origins of symbols can be indicated.

spec LIST_SET_1 [**sort** *Elem*] =
LIST_SELECTORS [**sort** *Elem*] **with** *empty*, *cons*
and GENERATED_SET [**sort** *Elem*] **with** *empty*, {__}, __ ∪ __
then **op** *elements_of __* : *List* → *Set*
∀*e* : *Elem*; *L* : *List*
- *elements_of empty* = *empty*
- *elements_of cons*(*e*, *L*) = {*e*} ∪ *elements_of L*

end

Since, as explained above, '**with** *empty*, *cons*' means '**with** *empty* ↦ *empty*, *cons* ↦ *cons*', identity renaming can be used just to emphasize the

fact that a given specification exports some symbols. This is illustrated in the specification LIST_SET_1 above, which is quite similar to LIST_SET, but for the fact that here we emphasize that LIST_SELECTORS exports in particular the operations *empty* and *cons*, and that GENERATED_SET exports in particular the operations *empty*, '{_}', and '∪'.

6.3 Hiding

Auxiliary symbols used in structured specifications can be hidden.

spec NATURAL_PARTIAL_SUBTRACTION_3 =
NATURAL_PARTIAL_SUBTRACTION_1 **hide** *suc*, *pre*
end

spec NATURAL_PARTIAL_SUBTRACTION_4 =
NATURAL_PARTIAL_SUBTRACTION_1
reveal *Nat*, *0*, *1*, _ + _, _ − _, _ * _, _ < _
end

When writing large specifications, it is quite frequent to rely on auxiliary operations (and predicates) to specify the operations (and predicates) of interest. Once these are defined, the auxiliary operations are no longer needed, and are better removed from the exported signature of the specification, which should include only the symbols that were required to be specified. This is the purpose of the **hide** construct.

Consider for instance the specification NATURAL_PARTIAL_SUBTRACTION_1 given in Chap. 4, p. 32. Once addition and subtraction are defined, the two basic operations *suc* and *pre* are no longer needed (since $suc(x)$ is more conveniently written $x + 1$, and similarly $pre(x)$ is expressed by $x - 1$), and can therefore be hidden. This is illustrated by the specification NATURAL_PARTIAL_SUBTRACTION_3 given above.

Depending on the relative proportion of symbols to be hidden or not, in some cases it may be more convenient to explicitly list the symbols to be exported by a specification rather than those to be hidden. The construct '**reveal**' can be used for that purpose, and '**hide**' and '**reveal**' are just two symmetric constructs to achieve the same effect. The use of '**reveal**' is illustrated in NATURAL_PARTIAL_SUBTRACTION_4 above, and the reader can convince himself that both NATURAL_PARTIAL_SUBTRACTION_3 and NATURAL_PARTIAL_SUBTRACTION_4 export exactly the same symbols. However, in this case the first specification is clearly more concise. A more convincing example of the use of '**reveal**' is provided by the following example:

spec PARTIAL_ORDER_2 = PARTIAL_ORDER **reveal pred** _ ≤ _ **end**

Similar rules to the ones explained for renaming apply to the **hide** and **reveal** constructs. One can qualify a symbol to be hidden or revealed by its kind (**sort**, **op** or **pred**), and by default, overloaded symbols are hidden (or revealed) simultaneously.

Note that hiding a sort entails hiding all the operations or predicates that use it in their profile. Similarly, revealing an operation or a predicate entails revealing all the sorts involved in its profile. For instance, in the specification PARTIAL_ORDER_2 above, revealing the predicate '$\leq$' entails revealing also the sort *Elem*.

As a consequence, hiding sorts should be used with care in the presence of subsorts. For instance, hiding the sort *Nat* in the specification POSITIVE given in Chap. 5, p. 41, leads to a specification of positive natural numbers with a sort *Pos* which has the expected carrier set, but without any operation or predicate available on it. Hiding the sort *Nat* in the specification POSITIVE_ARITHMETIC (see Chap. 5, p. 41) may seem more appropriate, but one should still note that the predicate '$<$' is no longer available in POSITIVE_ARITHMETIC **hide sort** *Nat*.

As a last remark, note that when convenient, **reveal** can be combined with a renaming of (some of) the exported symbols. For instance, in the above PARTIAL_ORDER_2 specification, we could have written '**reveal pred** $\_\_ \leq \_\_ \mapsto leq$' if, in addition to a restriction of the signature of PARTIAL_ORDER, we wanted to rename the infix predicate '$\_\_ \leq \_\_$' into a predicate *leq* with a functional notation.

6.4 Local Symbols

Auxiliary symbols can be made local when they do not need to be exported.

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **op** $insert : Elem \times List \to List$
$\forall e, e' : Elem;\ L : List$
- $insert(e, empty) = cons(e, empty)$
- $insert(e, cons(e', L)) = cons(e', insert(e, L))$ *when* $e' < e$ *else* $cons(e, cons(e', L))$

within **op** $order : List \to List$
$\forall e : Elem;\ L : List$
- $order(empty) = empty$
- $order(cons(e, L)) = insert(e, order(L))$

end

In many cases, auxiliary symbols are introduced for immediate use, and they do not need to be exported by the specification where they are declared. Then the best is to limit the scope of the declarations of such auxiliary symbols by using the '**local** ... **within** ...' construct. This is illustrated in the above specification LIST_ORDER, where the *insert* operation is introduced only for the purpose of the axiomatization of *order*. The operation *insert* has its scope limited to the part that follows '**within**', and is therefore not exported by the specification LIST_ORDER.

It is generally advisable to ensure that auxiliary symbols are declared in local parts of specifications.

Care is needed with local sort declarations.

spec LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **pred** __*is_sorted* : *List*
$\forall e, e' : Elem;\ L : List$
• *empty is_sorted*
• $cons(e, empty)$ *is_sorted*
• $cons(e, cons(e', L))$ *is_sorted* $\Leftrightarrow$
$(cons(e', L)$ *is_sorted* $\wedge \neg(e' < e))$
within **op** *order* : *List* $\rightarrow$ *List*
$\forall L : List \bullet order(L)$ *is_sorted*
end

The specification LIST_ORDER_SORTED above is a variant of the specification LIST_ORDER illustrating again the use of the '**local** ... **within** ...' construct – this time to declare an auxiliary predicate. (Actually, the two specifications are not equivalent, since LIST_ORDER_SORTED is much looser and only requires that $order(L)$ is a sorted list, but perhaps not with the same elements as L.)

spec WRONG_LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **pred** __*is_sorted* : *List*
sort $SortedList = \{L : List \bullet is\_sorted(L)\}$
$\forall e, e' : Elem;\ L : List$
• *empty is_sorted*
• $cons(e, empty)$ *is_sorted*
• $cons(e, cons(e', L))$ *is_sorted* $\Leftrightarrow$
$(cons(e', L)$ *is_sorted* $\wedge \neg(e' < e))$
within **op** *order* : *List* $\rightarrow$ *SortedList*
end

Note that the above specification WRONG_LIST_ORDER_SORTED, which may at first glance be considered as a slight variant of LIST_ORDER_SORTED, is *illegal*: *order* is exported by WRONG_LIST_ORDER_SORTED, and hence all sorts occurring in its profile should also be exported, which cannot be, since the sort *SortedList* is auxiliary. So, if the specifier really intends to insist that the result sort of *order* is *SortedList*, this subsort should be exported, as shown below.

spec LIST_ORDER_SORTED_2
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$(cons(e', L)\ is\_sorted \wedge \neg(e' < e))$
within **sort** $SortedList = \{L : List \bullet is\_sorted(L)\}$
op $order : List \to SortedList$
end

In fact the '**local** ... **within** ...' construct abbreviates a combination of extension and explicit hiding. In LIST_ORDER_SORTED_2 above, for instance, it abbreviates:

... **{** **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
• $empty\ is\_sorted$
• $cons(e, empty)\ is\_sorted$
• $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
$(cons(e', L)\ is\_sorted \wedge \neg(e' < e))$
then **sort** $SortedList = \{L : List \bullet is\_sorted(L)\}$
op $order : List \to SortedList$
}
hide $\_\_is\_sorted$

The main advantage of using the '**local** ... **within** ...' construct is that the symbol(s) to be hidden are left implicit. The convenience of this generally outweighs the danger of overlooking a locally-declared sort that is needed for the profile of an exported symbol. In any case, CASL allows both styles, and users can simply choose the one they prefer.

6.5 Named Specifications

Naming a specification allows its reuse.

It is in general advisable to define as many named specifications as felt appropriate, since this improves the reusability of specifications: a named specification can easily be reused by referring to its name.

Not only do the names serve as abbreviations when writing specifications, they also make it easy for readers to notice reuse. Moreover, when the name of a specification is aptly chosen, e.g., NATURAL_ARITHMETIC, readers may well be able to guess its signature – and perhaps even the specified axioms – from the name itself. (In Chap. 9, we shall see how named specifications and other items can be collected in libraries, and particular versions of them made available for use over the Internet.)

References to named specifications are particularly convenient for specifications structured using unions and extensions, where verbatim insertion of un-named specifications would tend to obscure the structure. When needed, the signature of a referenced specification can be adjusted through appropriate combinations of renaming and hiding (although this should not often be necessary, provided that auxiliary symbols are made local, as explained in the previous section).

7

Generic Specifications

Making a specification generic (when appropriate) improves its reusability.

As mentioned in the previous chapter, naming specifications is a good idea. In many cases, however, datatypes are naturally *generic*, having sorts, operations, and/or predicates that are deliberately left loosely specified, to be determined when the datatype is used. For instance, datatypes of lists and sets are generic regarding the sort of elements. *Generic specifications* allow the genericity of a datatype to be made explicit by declaring *parameters* when the specification is named: in the case of lists and sets, there is a single parameter that simply declares the sort *Elem*.[1] A fitting argument specification has to be provided for each parameter of a generic specification whenever it is referenced; this is called *instantiation* of the generic specification.

The aim of this chapter is to discuss and illustrate how to define generic specifications and instantiate them. We have seen plenty of simple examples of generic specifications and instantiations in the previous chapters. In more complicated cases, however, explicit fitting symbol maps may be required to determine the exact relationship between parameters and arguments in instantiations, and so-called *imports* should be separated from the bodies of generic specifications.

[1] Generic specifications are also useful to ensure loose coupling between several named specifications (of the same system), replacing an explicit extension by a parameter including only the necessary symbols and their required properties. This cannot however easily be illustrated in the framework of this User Manual.

7.1 Parameters and Instantiation

Parameters are arbitrary specifications.

Any specification, named or not, can be used as the parameter of a generic specification. Commonly, the parameter is a rather trivial specification consisting merely of a single sort declaration, as in most of the examples given in the previous chapters, e.g.:

spec LIST_SELECTORS [**sort** *Elem*] = ...

However, the parameter can also be a more complex, possibly structured, specification, as in:

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** __ < __] = ...

Recall that '**with**' requires the signature of the specification to include the listed symbols; here, in fact, the signature of TOTAL_ORDER does not contain any further symbols, so those are all the symbols that have to be supplied when instantiating LIST_ORDER.

In instantiations, the fitting of parameter symbols to identical argument symbols can be left implicit.

spec GENERIC_COMMUTATIVE_MONOID [**sort** *Elem*] =
GENERIC_MONOID [**sort** *Elem*]
then ...

When the parameter and the argument have symbols in common, these parameter symbols are implicitly taken to fit directly to the corresponding argument symbols. Thus it is *never* necessary to make explicit that a symbol is mapped identically. In the above example, for instance, the parameter specification of GENERIC_MONOID is exactly the same as the argument specification in its instantiation, so the fitting can be left implicit.

The fitting of parameter sorts to unique argument sorts can also be left implicit.

spec NAT_WORD =
GENERIC_MONOID [NATURAL]
end

When the argument specification has only a single sort, the fitting of all parameter sorts to that sort is obvious, and can again be left implicit. Of course, this does not apply the other way round: if the parameter has a single sort (which is often the case in practice) but the argument specification has more than one sort, the parameter sort could be mapped to any of the argument sorts, so the fitting symbol map has to be made explicit – except when the parameter sort is identical to one of the argument sorts, as previously explained.

Fitting symbol maps determine fitting morphisms.

A *(fitting) signature morphism* from the signature of the parameter part to the signature of the argument specification is automatically deduced, taking into account the explicitly specified fitting symbol map if any (the situation here is quite similar to a renaming, where a signature morphism is deduced from a symbol map). The instantiation is *defined* if all models of the argument specification, when reduced along the induced fitting signature morphism, provide models of the parameter part. In particular the symbols provided by the argument specification must have the properties, if any, specified in the parameter for their counterparts. When this is the case, we get not only a signature morphism, but also a *(fitting) specification morphism* from the argument specification to the parameter specification.

In the above NAT_WORD example, since the parameter of GENERIC_MONOID is trivial, it is obvious that the instantiation is defined.

The effect of the instantiation is to make the union of the argument specification and of the (non generic equivalent of the) generic specification, renamed according to the induced fitting signature morphism. In particular, a side-effect of the instantiation is to rename the symbols of the generic specification according to the fitting signature morphism induced by the instantiation. In our NAT_WORD example, the operation symbol $inj : Elem \to Monoid$ is renamed into $inj : Nat \to Monoid$, while the operation symbols '*1*' and '$*$' are left unchanged (as well as the sort *Monoid*). Thus, the specification NAT_WORD abbreviates the following specification:

NATURAL **and {** NON_GENERIC_MONOID **with** $Elem \mapsto Nat$ **}**.

When convenient, the instantiation can be completed by a renaming, as illustrated in the following variant of NAT_WORD:

spec NAT_WORD_1 =
GENERIC_MONOID [NATURAL]
with $Monoid \mapsto Nat\_Word$
end

Fitting of operations and predications can sometimes be left implicit too, and can imply fitting of sorts.

spec LIST_ORDER_NAT =
LIST_ORDER [NATURAL_ORDER]
end

In the above example, the fitting of both symbols of the parameter of LIST_ORDER (the sort *Elem* and the binary predicate __ < __) can be left implicit in the instantiation because the argument NATURAL_ORDER has only single symbols of the right kind. (The coincidence of the predicate symbol in the parameter and argument is irrelevant here.)

Fitting of function and predicate symbols can imply fitting of sorts. For instance, when a parameter predicate symbol is fitted to an argument predicate symbol whose profile involves different sorts, this implies that the parameter sorts involved have to be fitted to the corresponding sorts in the argument specification.

Note also that in the above example, checking the definedness of the instantiation corresponds to a non trivial proof obligation. The instantiation is defined since the predicate '<' provided by NATURAL_ORDER is indeed a total ordering relation, hence the properties required by TOTAL_ORDER are fulfilled, even if there is no syntactic correspondence between the axioms given in TOTAL_ORDER and those in NATURAL_ORDER.

As may be clear by now, the exact rules for when the fitting between parameter and argument symbols can be left implicit are quite sophisticated. It seems best to make the intended fitting explicit whenever it is not completely obvious, using the notation for fitting arguments illustrated in the following examples.

The intended fitting of the parameter symbols to the argument symbols may have to be specified explicitly.

spec NAT_WORD_2 =
GENERIC_MONOID [NATURAL_SUBSORTS **fit** *Elem* ↦ *Nat*]
end

The correspondence between the symbols required by the parameter and those provided by the argument specification can be made explicit using so-called *fitting symbol maps*. For instance, the above NAT_WORD_2 specification, which differs from NAT_WORD only regarding the presence of subsorts of *Nat*, is obtained as an instantiation of GENERIC_MONOID, fitting the parameter part '**sort** *Elem*' to the NATURAL_SUBSORTS specification. The mapping

between the parameter sort *Elem* and the sort *Nat* provided by NATURAL_SUBSORTS is described by the fitting symbol map '**fit** *Elem* ↦ *Nat*'.

A generic specification may have more than one parameter.

spec PAIR [**sort** *Elem1*] [**sort** *Elem2*] = ...

Using several parameters is merely a notational convenience, since they are equivalent to their union. For instance, the above PAIR specification could as well have been defined as follows:

spec PAIR_1 [**sorts** *Elem1*, *Elem2*] = ...

Note that writing:

spec HOMOGENEOUS_PAIR_1 [**sort** *Elem*] [**sort** *Elem*] =
free type *Pair* ::= *pair*(*first* : *Elem*; *second* : *Elem*)
end

merely defines pairs of values of the same sort, and HOMOGENEOUS_PAIR_1 is (equivalent to and) better defined as follows:

spec HOMOGENEOUS_PAIR [**sort** *Elem*] =
free type *Pair* ::= *pair*(*first* : *Elem*; *second* : *Elem*)
end

since the two parameters in HOMOGENEOUS_PAIR_1 are equivalent to just one '**sort** *Elem*' parameter.

From a methodological point of view, it is generally advisable to use as many parameters as convenient: the part of the specification that is intended to be specialized at instantiation time is better split into logically coherent units, each one corresponding to a parameter. Consider for instance:

spec SYMBOL_TABLE [**sort** *Symbol*] [**sort** *Attribute*] = ...

Here, using two parameters in SYMBOL_TABLE emphasizes that *Symbol* and *Attribute* are logically distinct entities which can be specialized as desired independently of each other.

Instantiation of generic specifications with several parameters is similar to the case of just one parameter.

spec PAIR_NATURAL_COLOUR =
PAIR [NATURAL_ARITHMETIC] [COLOUR **fit** *Elem2* ↦ *RGB*]
end

In the above example, the first parameter '**sort** *Elem1*' of PAIR is instantiated by NATURAL_ARITHMETIC, which exports only one sort *Nat*, hence no explicit fitting map is necessary. The second parameter '**sort** *Elem2*' of

PAIR is instantiated by COLOUR: in this case a fitting symbol map must be provided, since COLOUR exports two sorts, *RGB* and *CMYK*.

Using the specification PAIR_1 would require us to write:

spec PAIR_NATURAL_COLOUR_1 =
PAIR_1 [NATURAL_ARITHMETIC **and** COLOUR
fit *Elem1* ↦ *Nat*, *Elem2* ↦ *RGB*]
end

which clearly demonstrates the benefit of using two parameters as in PAIR instead of just one as in PAIR_1.

When parameters are trivial ones (i.e., just one sort), one can always avoid explicit fitting maps. Consider for instance the following alternative to PAIR_NATURAL_COLOUR:

spec PAIR_NATURAL_COLOUR_2 =
PAIR [**sort** *Nat*] [**sort** *RGB*]
and NATURAL_ARITHMETIC **and** COLOUR
end

This is particularly convenient when the argument specification exports several sorts. Compare for instance:

spec PAIR_POS =
HOMOGENEOUS_PAIR [**sort** *Pos*] **and** INTEGER_ARITHMETIC_1
end

and

spec PAIR_POS_1 =
HOMOGENEOUS_PAIR [INTEGER_ARITHMETIC_1 **fit** *Elem* ↦ *Pos*]
end

Note that the instantiation:
HOMOGENEOUS_PAIR_1 [NATURAL] [COLOUR **fit** *Elem* ↦ *RGB*]
is ill-formed, since it entails mapping the sort *Elem* to both *Nat* and *RGB*.

More generally, care is needed when the several parameters of a generic specification share some symbols, which in general is not advisable.

As a last remark, note that it is easy to specialize a generic specification with several parameters, as in the following version of SYMBOL_TABLE:

spec MY_SYMBOL_TABLE [**sort** *Symbol*] =
SYMBOL_TABLE [**sort** *Symbol*] [NATURAL_ARITHMETIC]
end

where we still have a parameter for symbols, but decide that the attributes are natural numbers.

Composition of generic specifications is expressed using instantiation.

spec SET_OF_LIST [**sort** *Elem*] =
GENERATED_SET [LIST_SELECTORS [**sort** *Elem*] **fit** *Elem* ↦ *List*]
end

The above generic specification SET_OF_LIST describes sets of lists of arbitrary elements, and is obtained by an instantiation of the generic specification GENERATED_SET, whose parameter '**sort** *Elem*' is instantiated by the specification LIST_SELECTORS, itself trivially instantiated. Since the (trivially instantiated) specification LIST_SELECTORS exports two sorts *Elem* and *List*, it is of course necessary to specify, in the instantiation of GENERATED_SET, the fitting map from the parameter sort *Elem* to the argument sort *List*.

Note that the following specification:

spec MISTAKE [**sort** *Elem*] =
GENERATED_SET [LIST_SELECTORS [**sort** *Elem*]]
end

just provides sets of arbitrary elements *and* lists of arbitrary elements, since the absence of explicit symbol map entails that the generic specification GENERATED_SET is instantiated by the identity fitting map *Elem* ↦ *Elem*.[2] If this was indeed the desired effect, one should rather write instead:

spec SET_AND_LIST [**sort** *Elem*] =
GENERATED_SET [**sort** *Elem*] **and** LIST_SELECTORS [**sort** *Elem*]
end

As illustrated by SET_OF_LIST, composition of generic specifications is fairly easy in CASL. Note however that this composition is achieved by means of appropriate instantiations (some possibly trivial), and that CASL does not provide higher-order genericity in its current version.

7.2 Compound Symbols

Compound sorts introduced by a generic specification get automatically renamed on instantiation, which avoids name clashes.

[2] However, the situation would be different if the parameter of GENERATED_SET would have been, e.g., '**sort** *Val*', since then the absence of an explicit fitting symbol map would have led to an ambiguity: in that case the specifier would have been forced to specify whether the sort *Val* is to be mapped to *Elem* or to *List*.

spec LIST_REV [**sort** *Elem*] =
  free type *List*[*Elem*] ::= *empty* |
    cons(*head* :? *Elem*; *tail* :? *List*[*Elem*])
  ops __ ++ __ : *List*[*Elem*] × *List*[*Elem*] → *List*[*Elem*],
    assoc, *unit empty*;
    reverse : *List*[*Elem*] → *List*[*Elem*]
  ∀ *e* : *Elem*; *L*, *L1*, *L2* : *List*[*Elem*]
  • *cons*(*e*, *L1*) ++ *L2* = *cons*(*e*, *L1* ++ *L2*)
  • *reverse*(*empty*) = *empty*
  • *reverse*(*cons*(*e*, *L*)) = *reverse*(*L*) ++ [*e*]
end

spec LIST_REV_NAT = LIST_REV [NATURAL] **end**

In the specification LIST_REV, we introduce a compound sort *List*[*Elem*] to denote lists (of arbitrary elements). When the specification LIST_REV is instantiated as in LIST_REV_NAT, the translation induced by the (implicit) fitting symbol map is applied to the component *Elem* also where it occurs in *List*[*Elem*], providing a sort *List*[*Nat*]. Thus, compound sorts can be seen as a convenient way of implicitly completing the instantiation by an appropriate renaming of the (compound) sorts introduced by the generic specification.

spec TWO_LISTS =
  LIST_REV [NATURAL] %% Provides the sort *List*[*Nat*]
and LIST_REV [INTEGER] %% Provides the sort *List*[*Int*]
end

Using a compound sort *List*[*Elem*] proves particularly useful in the above example TWO_LISTS, where we make the union of two distinct instantiations of LIST_REV. If we would have used an ordinary sort *List*, then an unintentional name clash would have arisen,[3] and we would have to complete each instantiation by an explicit renaming of the sort *List*.

Note that in the specification TWO_LISTS, we have two sorts *List*[*Nat*] and *List*[*Int*], hence two overloaded constants *empty* (one of each sort), which may need disambiguation when used in terms. (How to disambiguate terms is explained in Chap. 3, p. 13.)

Similarly, we have overloaded operation symbols *cons*, *head*, *tail*, ++, and *reverse*, but in general their context of use in terms will be enough to disambiguate which one is meant.

spec TWO_LISTS_1 =
  LIST_REV [INTEGER_ARITHMETIC_1 **fit** *Elem* ↦ *Nat*]
and LIST_REV [INTEGER_ARITHMETIC_1 **fit** *Elem* ↦ *Int*]
end

[3] And the specification TWO_LISTS would have been inconsistent, due to the same name, same thing principle and the fact that *List* is defined by a free type construct.

Since the specification INTEGER_ARITHMETIC_1 provides three sorts *Nat*, *Pos*, and *Int*, an explicit fitting symbol map is needed in the above instantiations, which provide the sorts *List*[*Nat*] and *List*[*Int*]. Note that the subsorting relation $Nat < Int$ does *not* entail $List[Nat] < List[Int]$, but of course this can be added if desired in an extension by a subsorting declaration.

Compound symbols can also be used for operations and predicates.

spec LIST_REV_ORDER [TOTAL_ORDER] =
LIST_REV [**sort** *Elem*]
then **local** **op** $insert : Elem \times List[Elem] \to List[Elem]$
$\forall e, e' : Elem;\ L : List[Elem]$
• $insert(e, empty) = cons(e, empty)$
• $insert(e, cons(e', L)) = cons(e', insert(e, L))$ *when* $e' < e$
else $cons(e, cons(e', L))$
within **op** $order[\_\_ < \_\_] : List[Elem] \to List[Elem]$
$\forall e : Elem;\ L : List[Elem]$
• $order[\_\_ < \_\_](empty) = empty$
• $order[\_\_ < \_\_](cons(e, L)) = insert(e, order[\_\_ < \_\_](L))$
end

spec LIST_REV_WITH_TWO_ORDERS =
LIST_REV_ORDER
[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, \_\_ < \_\_ \mapsto \_\_ < \_\_$]
%% Provides the sort *List*[*Int*] and the operation $order[\_\_ < \_\_]$
and LIST_REV_ORDER
[INTEGER_ARITHMETIC_ORDER **fit** $Elem \mapsto Int, \_\_ < \_\_ \mapsto \_\_ > \_\_$]
%% Provides the sort *List*[*Int*] and the operation $order[\_\_ > \_\_]$
then **%implies**
$\forall L : List[Int]$ • $order[\_\_ < \_\_](L) = reverse(order[\_\_ > \_\_](L))$
end

The above example illustrates the use of compound identifiers for operation symbols, and the same rules apply to predicate symbols. While in most cases using compound identifiers for sorts will be sufficient, in some situations it is also convenient to use them for operation or predicate symbols, as done here for $order[\_\_ < \_\_]$. When LIST_REV_ORDER is instantiated, not only the sort *List*[*Elem*] gets renamed (here, in *List*[*Int*]), but also the operation symbol $order[\_\_ < \_\_]$, according to the fitting symbol map corresponding to the instantiation. If we would not have used a compound identifier for the *order* operation, then an unintentional name clash would have arisen. Note that on the other hand we rely on the same name, same thing principle to ensure that the sorts *List*[*Int*] provided by each of the two instantiations are the same, which indeed is what we want for this example.

Of course we do not bother to use a compound identifier for the *insert* operation symbol. This operation being local, it is not exported by LIST_REV_ORDER and cannot be the source of unintentional name clashes in instantiations.

7.3 Parameters with 'Fixed' Parts

Tentative section title

> *Parameters should be distinguished from references to fixed specifications that are not intended to be instantiated.*

spec LIST_WEIGHTED_ELEM [**sort** *Elem* **op** *weight* : *Elem* → *Nat*]
given NATURAL_ARITHMETIC =
LIST_REV [**sort** *Elem*]
then **op** *weight* : *List*[*Elem*] → *Nat*
∀*e* : *Elem*; *L* : *List*[*Elem*]
- *weight*(*empty*) = *0*
- *weight*(*cons*(*e*, *L*)) = *weight*(*e*) + *weight*(*L*)

end

In the above example, we specialize lists of arbitrary elements to lists of elements equipped with a *weight* operation, which is then overloaded by a *weight* operation on lists. Therefore we specify that the generic specification LIST_WEIGHTED_ELEM has for parameter a specification extending the **given** specification NATURAL_ARITHMETIC by a sort *Elem* and an operation symbol *weight*. Thereby the intention is to emphasize the fact that only the sort *Elem* and the operation *weight* are intended to be specialized when the specification LIST_WEIGHTED_ELEM will be instantiated, and not the 'fixed part' NATURAL_ARITHMETIC. One could have written also:

spec LIST_WEIGHTED_ELEM
[NATURAL_ARITHMETIC **then sort** *Elem* **op** *weight* : *Elem* → *Nat*]
= ...

but the latter, which is correct, misses the essential distinction between the part which is intended to be specialized and the part which is 'fixed' (since, by definition, the parameter is the part which has to be specialized).

Note also that omitting the '**given** NATURAL_ARITHMETIC' clause would make the declaration:

spec LIST_WEIGHTED_ELEM [**sort** *Elem* **op** *weight* : *Elem* → *Nat*] = ...

ill-formed, since the sort *Nat* is not available.

To summarize, the **given** construct is useful to distinguish the 'true' parameter from the part which is 'fixed'. Both the parameter and the body of the generic specification extend what is provided by the **given** part, whose exported symbols are therefore available.

Argument specifications are always implicitly regarded as extension of the imports.

spec LIST_WEIGHTED_PAIR_NATURAL_COLOUR =
LIST_WEIGHTED_ELEM [PAIR_NATURAL_COLOUR **fit** *Elem* ↦ *Pair*, *weight* ↦ *first*]
end

The instantiation specified in LIST_WEIGHTED_PAIR_NATURAL_COLOUR is correct since the fitting map is the identity on all the symbols exported by the 'fixed' part NATURAL_ARITHMETIC (which happens here to be included in the argument specification PAIR_NATURAL_COLOUR). More generally, the argument specification is always regarded as an extension of the **given** 'fixed' part, and the fitting map should be the identity on all symbols this 'fixed' part exports. This is illustrated in the next example:

spec LIST_WEIGHTED_INSTANTIATED =
LIST_WEIGHTED_ELEM [**sort** *Value* **op** *weight* : *Value* → *Nat*]
end

Here we rely on a rather trivial instantiation (whose purpose is merely to illustrate our point) where the fitting symbol map can be omitted since no ambiguity arises and where the argument specification '**sort** *Value* **op** *weight* : *Value* → *Nat*' is well-formed because it is regarded as an extension of the **given** NATURAL_ARITHMETIC 'fixed' part of LIST_WEIGHTED_ELEM, which implies the sort *Nat* is available.

Imports are also useful to prevent ill-formed instantiations.

spec LIST_LENGTH [**sort** *Elem*] **given** NATURAL_ARITHMETIC =
LIST_REV [**sort** *Elem*]
then **op** *length* : *List*[*Elem*] → *Nat*
∀*e* : *Elem*; *L* : *List*[*Elem*]
• *length*(*empty*) = *0*
• *length*(*cons*(*e*, *L*)) = *length*(*L*) + *1*
then %implies
∀*L* : *List*[*Elem*] • *length*(*reverse*(*L*)) = *length*(*L*)
end

The specification LIST_LENGTH needs the sort *Nat* and the usual arithmetic operations provided by NATURAL_ARITHMETIC to specify the *length* operation. In this case it is clear that the **given** part has nothing to do with the (trivial) parameter of LIST_LENGTH. The reason to specify NATURAL_ARITHMETIC as a **given** 'fixed' part is that this will make instantiations of LIST_LENGTH similar to the following one well-formed.

spec LIST_LENGTH_NATURAL =
LIST_LENGTH [NATURAL_ARITHMETIC]
end

To understand this point, consider the following variant of LIST_LENGTH:

spec WRONG_LIST_LENGTH [**sort** *Elem*] =
NATURAL_ARITHMETIC **and** LIST_REV [**sort** *Elem*]
then ...
end

The specification WRONG_LIST_LENGTH is fine as long as one does not need to instantiate it with NATURAL_ARITHMETIC as argument specification. The instantiation WRONG_LIST_LENGTH [NATURAL_ARITHMETIC] is ill-formed since some symbols of the argument specification are shared with some symbols of the body (and not already occurring in the parameter) of the instantiated generic specification, which is illegal. Of course the same problem will occur with any argument specification which provides, e.g., the sort *Nat*.

As a consequence, one should remember two essential points. First, an instantiation is ill-formed as soon as there are some shared symbols between the argument specification and the body of the generic specification. Therefore, when designing a generic specification, it is generally advisable to turn auxiliary required specifications (such as NATURAL_ARITHMETIC for LIST_LENGTH) into **given** 'fixed' parts.

7.4 Views

Views are named fitting maps, and can be defined along with specifications.

view INTEGER_AS_TOTAL_ORDER :
TOTAL_ORDER **to** INTEGER_ARITHMETIC_ORDER =
$Elem \mapsto Int,\ \_\_ < \_\_ \mapsto \_\_ < \_\_$
end
view INTEGER_AS_REVERSE_TOTAL_ORDER :
TOTAL_ORDER **to** INTEGER_ARITHMETIC_ORDER =
$Elem \mapsto Int,\ \_\_ < \_\_ \mapsto \_\_ > \_\_$
end

spec LIST_REV_WITH_TWO_ORDERS_1 =
LIST_REV_ORDER [**view** INTEGER_AS_TOTAL_ORDER]
and LIST_REV_ORDER [**view** INTEGER_AS_REVERSE_TOTAL_ORDER]
then %implies
$\forall L : List[Int] \bullet order[\_\_ < \_\_](L) = reverse(order[\_\_ > \_\_](L))$
end

A view is nothing but a convenient way to name a specification morphism (induced by a symbol map) from a (parameter) specification to an (argument) specification. This proves particularly useful when the same instantiation (with the same fitting symbol map) is intended to be used several times: naming it once for all makes its reuse easier. Once a view is defined, as e.g. INTEGER_AS_TOTAL_ORDER above, it can be referenced in instantiations as in LIST_REV_ORDER [**view** INTEGER_AS_TOTAL_ORDER], where the keyword '**view**' makes it clear that the argument is not merely a named specification with an implicit fitting map, which would be written differently. The rules regarding the omission of 'evident' symbol maps in explicit fittings apply to named specification morphisms too.

Since a view is defined only when the given symbol map induces a specification morphism (i.e., all models of the target specification, when reduced along the signature morphism induced by the given symbol map, provide models of the source specification), it may be convenient to use views just to explicitly document the existence of some specification morphisms, even when these are not intended to be used in any instantiation. For instance, the view INTEGER_AS_TOTAL_ORDER can be seen as the requirement that INTEGER_ARITHMETIC_ORDER indeed specifies '$<$' to be a total ordering relation, and would therefore make sense even without being used later on in instantiations.

Views can also be generic.

view LIST_AS_MONOID [**sort** *Elem*] :
MONOID **to** LIST_REV [**sort** *Elem*] =
$Monoid \mapsto List[Elem],\ 1 \mapsto empty,\ \_\_ * \_\_ \mapsto \_\_ + + \_\_$
end

The above example illustrates again the use of a view as a 'proof obligation', since for the moment we do not have examples of generic specifications with MONOID as parameter.

8

Specifying Architectural Structure

Architectural specifications are meant for imposing structure on implementations, whereas specification-building operations only structure the text of specifications.

As explained in the previous chapters, the specification of a complex system may be fairly large and should be structured into coherent, easy to grasp, pieces. CASL provides a number of (powerful) specification-building operations to achieve this, as detailed in Chap. 6. Moreover, generic specifications, described in Chap. 7, provide pieces of specification that are easy to reuse in several contexts, where they can be adapted as desired by instantiating them.

Specification-building operations and generic specifications are useful *to structure the text of the specification* of the system under consideration. However, the models of a structured specification have no more structure than do those of a flat, unstructured, specification. Indeed, most examples given in the previous chapters could have been structured differently, with the same meaning (i.e., with the same models). Structured specifications are usually adequate at the requirements stage, where the focus is on the expected overall properties of the system under consideration.

In contrast, the aim of architectural specifications is *to prescribe the intended architecture of the implementation* of the system. Architectural specifications provide the means for specifying the various *components* from which the system will be built, and describing how these components have to be assembled to provide an implementation of the system of interest. At the same time, they allow the task of implementing a system to be split into independent, clearly-specified sub-tasks. Thus, architectural specifications are essential at the design stage, where the focus is on how to factor the implementation of the system into components glued together.

The aim of this chapter is to discuss and illustrate both the rôle of architectural specifications and how to express them in CASL.

The idea underlying architectural specifications is that eventually in the process of systematic development of modular software from specifications, components are implemented as software modules in some chosen programming language. However, this step is beyond the scope of specification formalisms, so in CASL and in this chapter we identify components with models (and with functions from models to models, in the case of generic components). The modular structure of the software under development, as described by an architectural specification, is therefore captured here simply as an explicit, structural way to build CASL models.

The illustrations in this chapter are artificially simple.

Architectural specifications, and more generally component-oriented approaches, are intended for relatively large systems. In this chapter, however, we have to rely on simple small examples to illustrate and explain CASL architectural specification concepts and constructs. After reading this chapter, the reader is encouraged to study App. C, which provides realistic examples of the use of architectural specifications.

The following structured specifications will be referred to later in this chapter when illustrating CASL architectural specifications:

spec NATURAL = **%{** As defined in Chap. 3, page 18 **}%**
spec COLOUR = **%{** As defined in Chap. 3, page 18 **}%**
spec NATURAL_ARITHMETIC = **%{** As defined in Chap. 3, page 20 **}%**

spec ELEM = **sort** *Elem*

spec CONT [ELEM] =
%{ Similar to GENERATED_CONTAINER in Chap. 3, page 15,
but with a compound sort $Cont[Elem]$ **}%**
generated type $Cont[Elem] ::= empty \mid insert(Elem;\ Cont[Elem])$
pred $\_\_is\_in\_\_ : Elem \times Cont[Elem]$
$\forall e, e' : Elem;\ C : Cont[Elem]$
- $\neg(e\ is\_in\ empty)$
- $e\ is\_in\ insert(e', C) \Leftrightarrow (e = e' \vee e\ is\_in\ C)$

end

spec CONT_DEL [ELEM] =
CONT [ELEM]
then **op** $delete : Elem \times Cont[Elem] \to Cont[Elem]$
$\forall e, e' : Elem;\ C : Cont[Elem]$
- $e\ is\_in\ delete(e', C) \Leftrightarrow ((e\ is\_in\ C) \wedge \neg(e = e'))$

end

spec REQ = CONT_DEL [NATURAL]

spec FLAT_REQ =
free type $Nat ::= 0 \mid suc(Nat)$
generated type $Cont[Elem] ::= empty \mid insert(Nat;\ Cont[Elem])$
pred $\_\_is\_in\_\_ : Nat \times Cont[Elem]$
op $delete : Nat \times Cont[Elem] \to Cont[Elem]$
$\forall x, y : Nat;\ C : Cont[Elem]$
- $\neg(x\ is\_in\ empty)$
- $x\ is\_in\ insert(y, C) \Leftrightarrow (x = y \vee x\ is\_in\ C)$
- $x\ is\_in\ delete(y, C) \Leftrightarrow ((x\ is\_in\ C) \wedge \neg(x = y))$

end

8.1 Architectural Specifications

Let's assume in the following that REQ describes our requirements about the system to be implemented. First, note that both REQ and FLAT_REQ have the same models, which illustrates our point about the fact that the CASL specification-building operations are merely facilities to structure the text of specifications into coherent units.

> *An architectural specification consists of a list of unit declarations, specifying the required components, followed by a* ***result*** *part indicating how these components are to be combined.*

arch spec SYSTEM =
units N : NATURAL;
C : CONT [NATURAL] **given** N;
D : CONT_DEL [NATURAL] **given** C
result D

The SYSTEM architectural specification is intended to prescribe a specific architecture for implementing the system specified by REQ.

The first part, introduced by the keyword **units**, indicates that we require the implementation of our system to be made of three components N, C, and D. The second part, introduced by the keyword **result**, indicates that the component D provides the desired implementation.

Each component is provided with its specification. The line:

N : NATURAL

declares a component N specified by NATURAL, which means simply that N should be a model of NATURAL.

The line:

C : CONT [NATURAL] **given** N

declares a component C which, given the previously declared component N, provides a model of CONT [NATURAL]. It is essential to understand that the component C must *expand* the assumed component N into a model of CONT [NATURAL], which means that C reduced to the signature of NATURAL must be equal to N. This property reflects the fact that a software module is supposed to use what it is given exactly as supplied, without altering it.

Similarly, the line:

D : CONT_DEL [NATURAL] **given** C

declares a component D which, given the component C, expands it into a model of CONT_DEL [NATURAL].

The final **result** is therefore simply D. (More complex examples of **result** expressions will be illustrated in examples below.)

As in the rest of CASL, visibility is linear in architectural specifications, meaning that any component must be declared before being used (e.g., the component N should be declared before being referred to by '**given** N' in the declaration of the component C in the architectural specification SYSTEM). Component names (such as N, C, and D in SYSTEM) are *local* to the architectural specification where they are declared, and are not visible outside it.

There can be several distinct architectural choices for the same requirements specification.

For instance, the following architectural specification corresponds to a different architectural choice for implementing our REQ specification.

arch spec SYSTEM_1 =
units N : NATURAL;
CD : CONT_DEL [NATURAL] **given** N
result CD

The architectural specifications SYSTEM and SYSTEM_1 both provide models of REQ. However, the former insists on an implementation made of three components, while the latter insists on an implementation made of two components.

Each unit declaration listed in an architectural specification corresponds to a separate implementation task, independent from the other ones.

For instance, in the architectural specification SYSTEM, the task of providing a component D expanding C and implementing CONT_DEL [NATURAL]

is independent from the tasks of providing implementations N of NATURAL and C of CONT [NATURAL]. Hence, when providing the component D, one cannot make any further assumption on how the component C is (or will be) implemented, besides what is expressly ensured by its specification.

To understand this, let us consider again the requirements specification REQ (or its variant FLAT_REQ). Among its models, there is one where containers are implemented by lists without repetitions, and in this model we can choose to implement *delete* by just removing the first found occurrence of the element to be deleted. In this model, however, we rely on the knowledge about the implementation of containers to decide how to implement *delete* – which is fine, since both are simultaneously implemented in the same model. In contrast, in the architectural specification SYSTEM, we request that containers are to be implemented in the component C while *delete* is to be provided by a separate component D. Imposing that the component D can be developed independently of the component C means that for D it is no longer possible to implement *delete* as sketched above, since this specific implementation choice may not be compatible with an independently chosen realization for C (where containers may be implemented by bags, for instance). In the case of the architectural specification SYSTEM_1, since both containers and the *delete* operation are implemented in the same component CD, we can of course decide to implement containers by lists without repetitions and *delete* as sketched above.

Thus the component D should expand *any* given implementation C of CONT [NATURAL] and provide an implementation of CONT_DEL [NATURAL], which is tantamount to providing a *generic* implementation G of CONT_DEL [NATURAL] which takes the particular implementation of CONT [NATURAL] as a parameter to be expanded. Then we obtain D by simply applying G to C.

Genericity here arises from the independence of the developments of C and D, rather than from the desire to build multiple implementations of CONT_DEL [NATURAL] using different implementations of CONT [NATURAL]. This is reflected by the fact that G is left implicit in the architectural specification SYSTEM.

A unit can be implemented only if its specification is a persistent extension of the specifications of its given units.

For instance, the component D can exist only if the specification CONT_DEL [NATURAL] is a persistent extension of CONT [NATURAL], i.e., if any model of the latter specification can be expanded into a model of the former one, which is indeed the case here. Similarly, the component C can exist since CONT [NATURAL] is a persistent extension of NATURAL.

Consider now the following variant of CONT_DEL and the associated variant of the architectural specification SYSTEM:

spec CONT_DEL_V [ELEM] =
CONT [ELEM]
then **op** $delete : Elem \times Cont[Elem] \to Cont[Elem]$
$\forall e, e' : Elem;\ C : Cont[Elem]$
- $delete(e, empty) = empty$
- $delete(e, insert(e', C)) = C\ when\ e = e'$
$else\ insert(e', delete(e, C))$
- $e\ is\_in\ delete(e', C) \Leftrightarrow ((e\ is\_in\ C) \wedge \neg(e = e'))$

end

arch spec INCONSISTENT =
units N : NATURAL;
C : CONT [NATURAL] **given** N;
D : CONT_DEL_V [NATURAL] **given** C
result D

The specification CONT_DEL_V [NATURAL] is consistent (has some models), but is not a persistent extension of CONT [NATURAL] (since, for instance, a model of CONT [NATURAL] where containers are realized by arbitrary lists, possibly with repetitions, cannot be expanded into a model of CONT_DEL_V [NATURAL]). As a consequence, in the architectural specification INCONSISTENT, the specification of the component D is inconsistent, since no component can expand *all* implementations C of CONT [NATURAL] into models of CONT_DEL_V [NATURAL]. The architectural specification INCONSISTENT is therefore itself inconsistent.

To summarize, architectural specifications not only prescribe the intended architecture of the implementation of the system, but they also ensure that the specified components can be developed independently of each other (which imposes a certain degree of genericity for these components).

8.2 Explicit Generic Components

Genericity of components can be made explicit in architectural specifications.

arch spec SYSTEM_G =
units N : NATURAL;
F : NATURAL $\to$ CONT [NATURAL];
G : CONT [NATURAL] $\to$ CONT_DEL [NATURAL]
result $G\,[F\,[N]]$

The architectural specification SYSTEM_G is a variant of SYSTEM; here we choose to specify the second and third components as explicit *generic components*.

The line:

F : NATURAL → CONT [NATURAL]

declares a generic component F. Given any component implementing (i.e., model of) NATURAL, F should expand it into an implementation of CONT [NATURAL]. The models of the generic-component specification NATURAL → CONT [NATURAL] are functions that map any model of NATURAL to a model of CONT [NATURAL]. These functions are required to be *persistent*, meaning that the result model expands the argument model.

The third component G is also specified as a generic component: given any implementation of CONT [NATURAL], G should expand it into an implementation of CONT_DEL [NATURAL].

Hence the whole system is obtained by the composition of applications $G[F[N]]$, as described in the **result** part. In CASL, such compositions are called *unit terms*.

The component C of SYSTEM corresponds to the application $F[N]$ in SYSTEM_G, and similarly the component D in SYSTEM corresponds to $G[C]$, i.e., to $G[F[N]]$ in SYSTEM_G.

A specification of the form *SP1* → *SP2* defines generic components *GC* that should always expand their argument into a model of the target specification. This makes only sense as long as the signature of the target specification contains the signature of *SP1*. This is why in CASL, *SP2* is always considered as an implicit extension of *SP1*, and *SP1* → *SP2* abbreviates *SP1* → **{** *SP1* **then** *SP2* **}**.[1] Moreover, since the generic component *GC* should expand *any* model of *SP1*, the specification *SP1* → *SP2* is *consistent* (i.e., has some models) if and only if the specification *SP1* **then** *SP2* is a persistent extension of *SP1*. Forgetting this fact is a potential source of inconsistent specifications of generic components in architectural specifications. For instance, the specification CONT [NATURAL] → CONT_DEL_V [NATURAL] is inconsistent, for the reasons explained at the end of the previous section.

A generic component may be applied to an argument richer than required by its specification.

arch spec SYSTEM_V =
units *NA* : NATURAL_ARITHMETIC;
F : NATURAL → CONT [NATURAL];
G : CONT [NATURAL] → CONT_DEL [NATURAL]
result $G[F[NA]]$

The above architectural specification SYSTEM_V is a variant of SYSTEM_G. Here we require a component *NA* implementing the specification NATURAL_ARITHMETIC, instead of a component *N* implementing NATURAL as

[1] When *SP2* is already defined as an extension of *SP1*, as it is the case for instance here for CONT_DEL [NATURAL], *SP2* is equivalent to *SP1* **then** *SP2*.

in SYSTEM_G (perhaps because we know that such a component is already available in some collection of previously-implemented components.)

The generic component F requires a component fulfilling the specification NATURAL, but can of course be applied to a richer argument, as in $F[NA]$. A similar reasoning applies to G.

More generally, a generic component can be applied to any component (or to any unit term) that can be reduced along some morphism to an argument of the required 'type' (i.e., to a model of the required specification). When necessary, a fitting symbol map can be used to describe the correspondence between the symbols provided by the argument and those expected by the generic component. We do not detail here the technicalities related to these fitting symbol maps, since they are quite similar to those used in instantiations of generic specifications and the notations are the same.

As a last remark, note that, similarly to what happens when instantiating a generic specification by an argument specification, when a generic component is applied to an argument richer than required, the extra symbols are kept in the result. Hence the **result** of the above architectural specification SYSTEM_V contains also the interpretations of the arithmetic and ordering operations on natural numbers, as they are provided by the component NA. This means in particular that the implementation described by SYSTEM_V has a larger signature than the one described by SYSTEM_G.

Specifications of components can be named for further reuse.

unit spec CONT_COMP = ELEM → CONT [ELEM]

unit spec DEL_COMP = CONT [ELEM] → CONT_DEL [ELEM]

arch spec SYSTEM_G1 =
units N : NATURAL;
F : CONT_COMP;
G : DEL_COMP
result $G[F[N]]$

In the above example, we name CONT_COMP the specification (of generic components) ELEM → CONT [ELEM]. Similarly, we name DEL_COMP the specification CONT [ELEM] → CONT_DEL [ELEM]. Then both named specifications can be reused in the architectural specification SYSTEM_G1 which is essentially the same as the architectural specification SYSTEM_G.

Generic specifications naturally give rise to specifications of generic components, which can be named for later reuse, as illustrated above by CONT_COMP. However, the reader should not confuse a generic specification (which is nothing else than a piece of specification that can easily be adapted by instantiation) with the corresponding specification of a generic component: the

latter cannot be instantiated, it is the specified generic component which gets *applied* to suitable components.

Persistent extensions of the form '**spec** *SP2* = *SP1* **then** *SP*' also naturally give rise to specifications of generic components of the form $SP1 \rightarrow SP2$, as illustrated by DEL_COMP.

In the architectural specification SYSTEM_G1, we use again the fact that the generic component F can be applied to richer arguments than models of ELEM (and similarly for G). Since ELEM is more general (has more models) than NATURAL, there are potentially fewer possibilities for implementing the generic component specified by CONT_COMP (which should be compatible with any model of ELEM) than there are for implementing the generic component specified by NATURAL → CONT [NATURAL] (which only needs to be compatible with models of NATURAL; a similar argument holds for DEL_COMP). As a consequence, the variant SYSTEM_G1 is not equivalent to the architectural specification SYSTEM_G.

So far we have always used named (structured) specifications to specify components. Un-named specifications can be used as well, as illustrated below.

unit spec DEL_COMP1 =
CONT [ELEM] → { **op** $delete : Elem \times Cont[Elem] \rightarrow Cont[Elem]$
$\forall e, e' : Elem;\ C : Cont[Elem]$
• $e\ is\_in\ delete(e', C) \Leftrightarrow$
$((e\ is\_in\ C) \wedge \neg(e = e'))$ }
end

DEL_COMP1 is a variant of DEL_COMP, where for the sake of the example we directly specify the *delete* operation instead of referring to the named specification CONT_DEL. Remember that in a specification of a generic component of the form $SP1 \rightarrow SP2$, *SP2* is always considered as an implicit extension of *SP1*, which explains why the above example is well-formed.

A generic component may be applied more than once in the same architectural specification.

arch spec OTHER_SYSTEM =
units N : NATURAL;
C : COLOUR;
F : CONT_COMP
result $F\,[N]$ **and** $F\,[C$ **fit** $Elem \mapsto RGB]$

The above architectural specification requires a component N specified by NATURAL, a component C specified by COLOUR, and a generic component F specified by CONT_COMP. Then, as described by the **result** part, the desired system is obtained by applying F to N, applying F to C (in this case, an explicit fitting symbol map is necessary, since COLOUR exports two sorts

RGB and *CMYK*). Finally both application results are combined, which is expressed by **and**.

Apart from **free**, all specification-building operations for structured specifications have natural counterparts at the level of components, which are expressed using the same keywords.[2] The reader should remember that specification-building operations work with specifications defining classes of models (e.g., union of specifications, denoted by **and**), while in architectural specifications we work with individual models (corresponding to components, as is the case here in OTHER_SYSTEM where **and** denotes the combination of two components.)

Similarly to union, renaming and hiding have natural counterparts at the level of components. For instance, remember that the implementation described by SYSTEM_V has a larger signature than the implementation described by SYSTEM_G. It is however easy to modify the **result** part of SYSTEM_V if what we really want is an implementation with the same signature as the implementation described by SYSTEM_G: one has just to hide the extra symbols resulting from the component *NA* as follows:

result $G\,[F\,[NA]]$ **hide** $\_\_ < \_\_,\ 1,\ \_\_ + \_\_,\ \_\_ * \_\_$
or:
result $G\,[F\,[NA$ **hide** $\_\_ < \_\_,\ 1,\ \_\_ + \_\_,\ \_\_ * \_\_]]$

Symbol maps used in renaming and hiding at the level of components follow the same rules as symbol maps used in renaming and hiding at the level of structured specifications (see Chap. 6).

Several applications of the same generic component is different from applications of several generic components with similar specifications.

arch spec OTHER_SYSTEM_1 =
units N : NATURAL;
C : COLOUR;
FN : NATURAL $\rightarrow$ CONT [NATURAL];
FC : COLOUR $\rightarrow$ CONT [COLOUR **fit** *Elem* $\mapsto$ *RGB*]
result $FN\,[N]$ **and** $FC\,[C]$

The above architectural specification OTHER_SYSTEM_1 is a variant of OTHER_SYSTEM. However, in OTHER_SYSTEM, we insist on choosing one implementation for containers in the generic component F, and then we apply it twice, first to a component N implementing NATURAL, and then to a component C implementing COLOUR. In contrast, in OTHER_SYSTEM_1, we may

[2] The situation is however a bit different with specification extensions, which lead to specifications of generic components, as explained above, or to specifications of components expanding a **given** component, as illustrated in the previous section.

choose two different implementations for containers, one for containers of natural numbers in the component *FN* and another one for containers of colours in the component *FC*.

The architectural specifications OTHER_SYSTEM and OTHER_SYSTEM_1 are therefore similar but clearly different. Neither is better than the other: each corresponds to a different architectural decision, and selecting one rather than the other is a matter of architectural design. Components that are more widely reusable tend to have less-efficient implementations, in general.

Generic components may have more than one argument.

unit spec SET_COMP = ELEM → GENERATED_SET [ELEM]

spec CONT2SET [ELEM] =
CONT [ELEM] **and** GENERATED_SET [ELEM]
then **op** *elements_of__* : *Cont*[*Elem*] → *Set*
∀ *e* : *Elem*; *C* : *Cont*[*Elem*]
• *elements_of empty* = *empty*
• *elements_of insert*(*e*, *C*) = {*e*} ∪ *elements_of C*
end

arch spec ARCH_CONT2SET_NAT =
units *N* : NATURAL;
C : CONT_COMP;
S : SET_COMP;
F : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
result *F* [*C* [*N*]] [*S* [*N*]]

The architectural specification ARCH_CONT2SET_NAT requires a component *N* implementing NATURAL, a generic component *C* implementing CONT_COMP, i.e., containers, and a generic component *S* implementing SET_COMP, i.e., sets. Then it further requires a generic component *F* that, given *any pair of compatible* models *X* of CONT [ELEM] and *Y* of GENERATED_SET [ELEM], expands them into a model of CONT2SET [ELEM].

Models *X* and *Y* are said to be *compatible* if they share a common interpretation for all symbols they have in common.[3] Here the only symbol they have in common is the sort *Elem*, so the compatibility condition means that *X* and *Y* have the same carrier set for *Elem*. Compatibility is a natural condition, since it is obviously necessary that *X* and *Y* have a common interpretation of their common symbols, otherwise they cannot be both expandable to the same more complex component.

[3] The compatibility condition is a bit more subtle in the presence of subsorts, for more details see [Mos03, Secs. III:3.1.2, III:5.1].

The **result** is then obtained by applying F to the pair obtained by applying C to N and S to N. Here the pair of arguments $C\,[N]$ and $S\,[N]$ are obviously compatible, since their common symbols (the sort *Nat* equipped with the operations *0* and *suc*) all come from the *same* component N which provides their interpretation, which is expanded (hence cannot be modified) in $C\,[N]$ and in $S\,[N]$, thus compatibility is guaranteed.

Open systems can be described by architectural specifications using generic unit expressions in the ***result*** *part.*

arch spec ARCH_CONT2SET =
units C : CONT_COMP;
S : SET_COMP;
F : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
result λX : ELEM • $F\,[C\,[X]]\,[S\,[X]]$

arch spec ARCH_CONT2SET_USED =
units N : NATURAL;
CSF : **arch spec** ARCH_CONT2SET
result *CSF* $[N]$

So far our example architectural specifications have described 'closed', stand-alone systems where all components necessary to built the desired system were declared in the architectural specification of interest. In CASL, it is however possible to describe 'open' systems, i.e., systems made of some components that would require further components to provide a 'closed' system. This is illustrated by the architectural specification ARCH_CONT2SET which describes a system with a generic component C implementing containers, a generic component S implementing sets, and a generic component F that expands them to provide an implementation of the operation *elements_of*. The **result** part is therefore a generic structured component, i.e., an 'open' system, which, given any component X implementing ELEM, provides a system built by applying F to the pair made of the applications of C to X and of S to X.

As illustrated by ARCH_CONT2SET_USED, we can then describe a 'closed' system made of a component N implementing NATURAL, and of an 'open' system *CSF* specified by ARCH_CONT2SET, which is then applied to N in the **result** part.

8.3 Writing Meaningful Architectural Specifications

In the previous sections we have already pointed out potential sources of inconsistent specifications of components. Another issue which deserves some attention when designing an architectural specification is *compatibility* between components (or, more generally, unit terms) that are to be combined together, either by **and**, or by fitting them to a generic component with multiple arguments.

> *In an architectural specification, it is advisable to ensure that any shared symbol between two components or unit terms that are to be combined can be traced to a single non-generic component.*

arch spec ARCH_CONT2SET_NAT_V =
units N : NATURAL;
C : CONT_COMP;
S : SET_COMP;
G : **{** CONT [ELEM] **and** GENERATED_SET [ELEM] **}**
→ CONT2SET [ELEM]
result $G\,[C\,[N]$ **and** $S\,[N]]$

The architectural specification ARCH_CONT2SET_NAT_V is a variant of ARCH_CONT2SET_NAT where, instead of declaring a generic component F with two arguments, we now declare a generic component G with a single argument, which must be a model of the specification **{** CONT [ELEM] **and** GENERATED_SET [ELEM] **}**, obtained as the union of the two (trivially instantiated) specifications of containers and sets.[4]

As a consequence, to obtain the desired system, in the **result** part we apply the generic component G to the combination (denoted by **and**) of C applied to N and of S applied to N. This combination makes sense only if both $C\,[N]$ and $S\,[N]$ share the same interpretation of their common symbols. Here their common symbols (the sort *Nat* equipped with the operations *0* and *suc*) all come from the *same* component N which provides their interpretation, which is expanded (hence cannot be modified) in $C\,[N]$ and in $S\,[N]$, thus compatibility is guaranteed.

There is a clear analogy here between the application of the generic component F with multiple arguments in ARCH_CONT2SET_NAT and the combination of $C\,[N]$ and $S\,[N]$ in ARCH_CONT2SET_NAT_V: in both cases the **result** is meaningful because we can trace shared symbols like the sort *Nat* and the operations *0* and *suc* to a single component N introducing them.

Let us emphasize again that compatibility is a natural requirement: since each unit declaration corresponds to a separate implementation task (and

[4] The braces '**{**' and '**}**' are not really necessary and are used to emphasize that G has a single argument.

hence each unit subterm to an independently developed subsystem), obviously the combination of components or unit terms makes sense only when some compatibility conditions are fulfilled.

Let us now consider an example where the compatibility condition is violated:

arch spec WRONG_ARCH_SPEC =
units CN : CONT [NATURAL];
SN : GENERATED_SET [NATURAL];
F : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
result $F\ [CN]\ [SN]$

The architectural specification WRONG_ARCH_SPEC is a variant of ARCH_CONT2SET_NAT where, instead of requiring a component N implementing NATURAL and two generic components implementing containers and sets respectively, we just require a component CN implementing containers of natural numbers and a component SN implementing sets of natural numbers. However, then the application $F\ [CN]\ [SN]$ makes no sense since there is no way to ensure that the common symbols of CN and SN have the same interpretation. It may indeed be the case that natural numbers are interpreted in some way in CN and in a different way in SN, which makes the application of F impossible.

Let us now consider a more complex example:

arch spec BADLY_STRUCTURED_ARCH_SPEC =
units N : NATURAL;
A : NATURAL → NATURAL_ARITHMETIC;
C : CONT_COMP;
S : SET_COMP;
F : CONT [ELEM] × GENERATED_SET [ELEM]
→ CONT2SET [ELEM]
result $F\ [C\ [A\ [N]]]\ [S\ [A\ [N]]]$

The architectural specification BADLY_STRUCTURED_ARCH_SPEC is a variant of ARCH_CONT2SET_NAT where, in addition to the component N implementing NATURAL, we require a generic component A which is used to expand N into an implementation of NATURAL_ARITHMETIC. In the architectural specification ARCH_CONT2SET_NAT, the compatibility condition in the application $F\ [C\ [N]]\ [S\ [N]]$ was easy to discharge. Here, in the **result** unit term $F\ [C\ [A\ [N]]]\ [S\ [A\ [N]]]$ of BADLY_STRUCTURED_ARCH_SPEC, we apply F to the pair made of $C\ [A\ [N]]$ and $S\ [A\ [N]]$. In this case only a semantic analysis can ensure that these two arguments are compatible, since the common symbols cannot be traced to the same non-generic component, but only to two applications of the same generic component A to similar arguments.

It is advisable to use unit terms where compatibility can be checked by a simple static analysis. CASL provides additional constructs which make it easy to follow this recommendation, as explained below.

Auxiliary unit definitions or local unit definitions may be used to avoid repetition of generic unit applications.

```
arch spec WELL_STRUCTURED_ARCH_SPEC =
units N  : NATURAL;
      A  : NATURAL → NATURAL_ARITHMETIC;
      AN = A [ N ];
      C  : CONT_COMP;
      S  : SET_COMP;
      F  : CONT [ ELEM ] × GENERATED_SET [ ELEM ]
                  → CONT2SET [ ELEM ]
result F [C [AN]] [S [AN]]

arch spec ANOTHER_WELL_STRUCTURED_ARCH_SPEC =
units N : NATURAL;
      A : NATURAL → NATURAL_ARITHMETIC;
      C : CONT_COMP;
      S : SET_COMP;
      F : CONT [ ELEM ] × GENERATED_SET [ ELEM ]
                  → CONT2SET [ ELEM ]
result local AN = A [N] within F [C [AN]] [S [AN]]
```

The problem illustrated in BADLY_STRUCTURED_ARCH_SPEC can be fixed easily. An auxiliary unit definition may be used to avoid the repetition of some generic unit applications, such as '$AN = A\,[N]$' in WELL_STRUCTURED_ARCH_SPEC. An alternative is to make the definition of AN local to the **result** unit term, as illustrated in ANOTHER_WELL_STRUCTURED_ARCH_SPEC. In both case common symbols can be traced to a non-generic unit, and compatibility can be checked by an easy static analysis.

9

Libraries

Libraries are named collections of named specifications.

In the foregoing chapters, we have seen many examples of named specifications, and of references to them in later specifications. This chapter explains how a collection of named specifications can itself be named, as a *library*. The creation of libraries facilitates the *reuse* of specifications. For practical applications, it is important to be able to reuse (at least) existing specifications of basic datatypes, such as those described in App. B.

Local libraries are self-contained.

A library is called *local* when it is self-contained, i.e., for each reference to a specification name in the library, the library includes a specification with that name. Local libraries might appear at first sight to be all that we need, but actually they provide poor support for reuse of specifications. The problem is that when a specification from one local library is reused in another, it has to be repeated *verbatim*. There is no formal link between the original specification and the copy, despite them having the same name: the names used in a library can be chosen freely, and different libraries could use the same name for completely different specifications.

Distributed libraries support reuse.

Distributed libraries allow duplication of specifications to be avoided altogether. Instead of making an explicit copy of a named specification from

one library for use in another, the second library merely indicates that the specification concerned can be *downloaded* from the first one.

To maximize reuse, each specification should be given only once, in a single library. In practice, however, libraries could be developed in parallel, without collaboration, and it might happen that a specification in one library is very similar to a (perhaps differently-named) specification in another library. When such duplication is noticed, and the specification concerned is appropriate for general use, the developers will be encouraged (but not required) to agree on using a common, uniquely-named specification, defined in only one of the libraries.

Different versions of the same library are distinguished by hierarchical version numbers.

In practice, specifications *evolve*, e.g., to provide further operations or predicates on the specified sorts, or to define new subsorts. The libraries containing the specifications can evolve too, by adding or removing named specifications. Without some form of version control, even a trifling change in one library might cause specifications in other libraries to become ill-formed, or affect their classes of models. CASL allows different versions of the same library to coexist (distinguishing them by hierarchical version numbers), and allows downloadings in a library to indicate that a particular version of another library is required.

Creation of new libraries is essential in connection with larger specification projects, and projects of any scale can benefit from reuse of specifications from existing libraries. The rest of this chapter illustrates the constructs used to specify local libraries, distributed libraries, and versions, and gives some advice on the organization of libraries.

9.1 Local Libraries

Local libraries are self-contained collections of specifications.

library USERMANUAL/EXAMPLES
...
spec NATURAL = ...
...
spec NATURAL_ORDER = NATURAL **then** ...
...

The collection of all the illustrative examples given in the foregoing chapters is self-contained, so it could be made into a local library and named UserManual/Examples, as outlined above. It would also be possible to divide our illustrative examples into local libraries according to a general classification of datatypes: ordering relations, numbers, lists, sets, etc. (with a further library for miscellaneous datatypes such as that of vehicles). This organization would facilitate browsing the illustrative specifications concerned with particular kinds of datatypes. However, some duplication of specifications would be required (e.g., natural numbers are needed in connection with both lists and vehicles).

The 'same name, same thing' principle of CASL applies only within specifications, and it is possible for a library to include alternative specifications for the same symbols (e.g., using different sets of axioms). However, when such alternative specifications are both extended (perhaps indirectly) in the same specification, the principle does apply, and inconsistency might then arise.

Thus in general, it is advisable for the developers of a library to respect the 'same name, same thing' principle when choosing symbols throughout the library. In any case, this is obviously helpful to those who might later browse the library. Alternative specifications for the same symbols should therefore be given in separate libraries.[1]

Specifications can refer to previous items in the same library.

library UserManual/Examples
...
spec Strict_Partial_Order = ...
...
spec Total_Order = Strict_Partial_Order **then** ...
...
spec Partial_Order = Strict_Partial_Order **then** ...
...

Although we may often regard libraries as *sets* of named specifications, they are actually *sequences*, and the order in which the specifications occur is significant.

Specification names have linear visibility: each specification can refer only to the names of the specifications that precede it. Thus a series of extensions has to be presented in a bottom-up fashion, starting with a specification that is entirely self-contained, containing no references to other specifications at all. Each specification name in a library has a unique defining occurrence, so overriding cannot arise. Extensions that do not refer to each other may be

[1] If we intended our comprehensive UserManual/Examples library for general use, we would remove all the illustrative alternative specifications.

given in any order (e.g., PARTIAL_ORDER above could be given after TOTAL_ORDER).

Mutual recursion between specifications is prohibited. When two specifications each make use of symbols declared in the other, the declarations of those symbols have to be duplicated, or moved to a preceding specification that can then be referenced by them both.

All kinds of named specifications can be included in libraries.

library USERMANUAL/EXAMPLES
...
spec STRICT_PARTIAL_ORDER = ...
...
spec GENERIC_MONOID [**sort** *Elem*] = ...
...
view INTEGER_AS_TOTAL_ORDER : ...
...
view LIST_AS_MONOID [**sort** *Elem*] : ...
...
arch spec SYSTEM = ...
...
unit spec CONT_COMP = ...
...

Items in libraries can be any kind of named specification, as illustrated above: simple named specifications, generic specifications, named view definitions, generic view definitions, and architectural and unit specifications. We shall henceforth refer to them generally as *library items*.

Libraries themselves never include anonymous specifications, such as declarations of sorts and operations. Moreover, the symbols declared by a library item are *not* automatically available for use in subsequent items: an explicit reference to the name of the library item is required to "import" the item.

Technically, each library item is said to be *closed*, being interpreted without any pre-declared symbols at all. This facilitates reordering library items, and (more importantly) copying items between libraries.

Display, parsing, and literal syntax annotations apply to entire libraries.

library UserManual/Examples
...
%display __<=__ %LATEX __ ≤ __
%display __>=__ %LATEX __ ≥ __
%display __union__ %LATEX __ ∪ __

This is consistent with the display annotations given in the basic libraries.

%prec {__+__, __−__} < {__ * __}
%left_assoc __+__, __ * __
...
spec Strict_Partial_Order = ...
...
spec Partial_Order = Strict_Partial_Order **then** ... ≤ ...
...
spec Generated_Set [**sort** *Elem*] = ... ∪ ...
...
spec Integer_Arithmetic_Order = ... ≤ ... ≥ ...
...

Annotations affecting the way terms are written or displayed apply only to an entire library, and have to be collected at the beginning of the library. These annotations include display and precedence annotations, illustrated above.

Recall that various reserved words and symbols in CASL specifications are input in ASCII, but displayed as mathematical signs (e.g., universal quantification is input as '`forall`', and displayed as '∀' when this sign is available in the current display format). *Display annotations* provide analogous flexibility for declared symbols. For example, the display annotations illustrated above determine how infix symbols input as `<=`, `>=`, and `union` are displayed when using LaTeX to format the specification. Note that a display annotation applies to all occurrences of the input symbol in the library, regardless of overloading.

Display annotations can give alternative displays for different formats: apart from LaTeX, both RTF and HTML are presently supported. The display of the annotation itself shows only the input syntax of the symbol and the result produced by the current formatter. The input form of one of the above annotations might be as follows:

Should HTML be replaced by XHTML?

```
%display __union__ %HTML __<sup>U</sup> %LATEX __\cup__
```

When no display annotation is given for a particular format, the input format itself is displayed. Thus the symbol displayed as ∪ in the present LaTeX version of this User Manual would be displayed as *union* in an RTF version, unless the above annotation were to be extended with an RTF part.

Parsing annotations allow omission of grouping parentheses when terms are input. A single annotation can indicate the relative precedence or the

associativity (left or right) of a group of operation symbols. The precedence annotation for infix arithmetic operations given above allows a term such as $a + (b * c)$ to be input (and hence also displayed) as $a + b * c$. The left-associativity annotation for $+$ and $*$ allows $(a+b)+c$ to be input as $a+b+c$, and similarly for $*$; but the parentheses cannot be omitted in $(a+b)-c$ (not even if $\_\_-\_\_$ were to be included in the same left-associativity annotation).

When an operation symbol is declared with the associativity *attribute assoc*, a (left-)associativity *annotation* for that symbol is provided automatically. Thus in practice, explicit associativity annotations are needed only for non-associative operations such as subtraction and division.

Libraries and library items can have author and date annotations.

library UserManual/Examples
...
%authors(Michel Bidoit `<bidoit@lsv.ens-cachan.fr>`,
Peter D. Mosses `<pdmosses@brics.dk>` **)%**
%dates 15 Oct 2003, 1 Apr 2000
...
spec Strict_Partial_Order = ...
...
%authors Michel Bidoit `<bidoit@lsv.ens-cachan.fr>`
%dates 10 July 2003
spec Integer_Arithmetic_Order =
...

An *author annotation* at the beginning of a library indicates the collective authorship of the entire library; one preceding an individual library item indicates its specific authorship.

A *date annotation* at the beginning of a library should indicate the release date of the current version of the library. It may also give the release dates of previous major versions, including that of the original version. A date annotation on an individual library item should indicate when that item was last changed, and (optionally) the dates of previous changes.

9.2 Distributed Libraries

Libraries can be installed on the Internet for remote access.

library BASIC/NUMBERS
...
%left_assoc __@@__
%number __@@__
%floating __:::__, __E__
%prec {__E__} < {__:::__}
...
spec NAT =
free type $Nat ::= 0 \mid suc(Nat)$
...
ops $1 : Nat = suc(0); \ldots; 9 : Nat = suc(8);$
$\_\_@@\_\_(m, n : Nat) : Nat = (m * suc(9)) + n$
...
...
spec INT = NAT **then** ...
...
spec RAT = INT **then** ...
...
spec DECIMALFRACTION = RAT **then**
...
ops $\_\_:::\_\_ : Nat \times Nat \to Rat;$
$\_\_E\_\_ : Rat \times Int \to Rat$
...

The above example is an extract from one of the CASL libraries of basic datatypes, described in App. B and available on the Internet. It illustrates the overall structure of a library intended for general use, as well as some helpful annotations concerning literal syntax for numbers, which are explained below.

CASL libraries are identified by hierarchical path names. For instance, all the CASL libraries of basic datatypes have names starting with 'BASIC/', and path names starting with 'CASL/' are reserved for libraries connected with the CASL language itself (e.g., the specification of the abstract syntax of CASL in CASL).

COFI is to maintain a register of validated libraries. The validation of a library ensures not only that it is well-formed, but also that the specifications and views are consistent, and that all proof obligations (corresponding to semantic annotations in the library) have been verified. Moreover, some methodological guidelines concerning the style of names have been provided, to facilitate the combined use of specifications taken from different libraries (see [Mos03, Part VI] for further details).

Validated libraries can be located via the CoFI web pages, and are to be mirrored at several sites, to ensure their accessibility.

It is likely that new versions of existing libraries will be produced, e.g., providing further operations whose usefulness was not realized beforehand. Although the assignment and use of library version numbers allows users to protect their specifications from changes due to new versions (see Sec. 9.3), at least the names used in an installed library should not change much between versions.

Libraries should include appropriate annotations. In particular, parsing and display annotations can be provided. The above example illustrates a further kind of annotation, used to provide *literal syntax* for numbers in CASL. The effect of the annotation is that, when using the appropriate specifications from the library BASIC/NUMBERS, conventional decimal notation can be used for integers and decimal fractions, e.g., *42*, *2.718*, *10E−12*.

Libraries can include items downloaded from other libraries.

library BASIC/STRUCTUREDDATATYPES
...
from BASIC/NUMBERS **get** NAT, INT
...
spec LIST [**sort** *Elem*] **given** NAT = ...
...
spec ARRAY ... **given** INT = ...
...

Individual specifications and other items can be *downloaded* from other libraries. For example, as indicated above, the library BASIC/STRUCTUREDDATATYPES does not itself provide the specifications of natural numbers and integers that are needed in the specifications of LIST and ARRAY, but instead downloads NAT and INT from the BASIC/NUMBERS library.

The names of the items to be downloaded from a library have to be listed explicitly: one cannot request the downloading of all the items that happen to be provided by a library. However, although the name of each item provided by a downloading is always explicit, no indication is given of its kind (i.e., whether it is an ordinary, generic, or architectural specification, or a view) nor of what symbols it declares. Thus the well-formedness of a library depends on what items are actually downloaded from other libraries.

The item '**from** ... **get** ...' above has the effect of downloading the specifications that are named NAT and INT in BASIC/NUMBERS, preserving their names. It is also possible to give downloaded specifications different names, e.g., to avoid clashes with names that are already in use locally:

from BASIC/NUMBERS **get** NAT ↦ NATURAL, INT ↦ INTEGER

Items that are referenced by downloaded items are *not* themselves automatically downloaded, e.g., downloading INT does not entail the downloading of NAT. This is because downloading involves the *semantics* of the named items, not their *text*. The semantics of INT consists of a signature and a class of models, and is a self-contained entity – recall from Chap. 8 that the models of a structured specification have no more structure than do those of a flat, unstructured specification. Thus downloading INT gives exactly the same result as if the reference to NAT in its text had been replaced by the text of NAT. For the same reason, the presence of another item with the name NAT in the current library makes no difference to the result of downloading INT. In terms of software packages, downloading from CASL libraries is analogous to installing from binaries, not from sources.

Downloading any item from another library B in a library A causes all the parsing and display annotations of B to be inserted at the beginning of A. (Conflicting annotations from different libraries are ignored altogether, and local annotations override conflicting downloaded annotations.) The copied annotations allow terms to be written and displayed in A in the same way as in B.

Substantial libraries of basic datatypes are already available.

The organization of the following libraries of basic datatypes is explained in App. B:

BASIC/NUMBERS: natural numbers, integers, and rationals.
BASIC/RELATIONSANDORDERS: reflexive, symmetric, and transitive relations, equivalence relations, partial and total orders, boolean algebras.
BASIC/ALGEBRA_1: monoids, groups, rings, integral domains, and fields.
BASIC/SIMPLEDATATYPES: booleans, characters.
BASIC/STRUCTUREDDATATYPES: sets, lists, strings, maps, bags, arrays, trees.
BASIC/GRAPHS: directed graphs, paths, reachability, connectedness, colourability, and planarity.
BASIC/ALGEBRA_II: monoid and group actions on a space, euclidean and factorial rings, polynomials, free monoids, and free commutative monoids.
BASIC/LINEARALGEBRA_I: vector spaces, bases, and matrices.
BASIC/LINEARALGEBRA_II: algebras over a field.
BASIC/MACHINENUMBERS: bounded subtypes of naturals and integers.

Libraries need not be registered for public access

library `http://www.brics.dk/~pdm/CASL/Test/Security`
...
from `http://casl:password@www.brics.dk/~pdm/CASL/RSA` **get** KEY
...
spec DECRYPT = KEY **then** ...
...

Libraries under development, and libraries provided for restricted groups of users, are named and accessed by their URLs. This allows the CASL library constructs to be fully exploited for libraries that are not yet – and perhaps never will be – registered for public access. Moreover, validation of libraries can be a demanding and time-consuming process, and getting a library approved and registered is appropriate only when it provides specifications that are likely to be found useful by persons not directly involved it its development.

The primary Internet access protocols HTTP and FTP both support password protection of specifications, and the insertion of usernames and passwords in URLs allows downloading between protected specifications. (With HTTP, the username and password can be unrelated to those used for the host file system.)

Does FTP also allow for artificial users?

9.3 Version control

Libraries always have version numbers

library BASIC/NUMBERS **version** 1.0
...
spec NAT = ...
...
spec INT = NAT **then** ...
...
spec RAT = INT **then** ...
...

As illustrated above, a library can be assigned an explicit version number, allowing it to be distinguished from previous and future versions of the same library. CASL allows conventional hierarchical version numbers, familiar from version numbers of software packages: the initial digits indicate a major version, digits after a dot indicate sub-versions, and digits after a further dot indicate patches to correct bugs. (Distinctions between alpha, beta, and other pre-release versions are not supported.)

The smallest version number is written simply '0', and can be omitted when specifying the initial version of a library. However, the first installed version of a library could have any version number at all. The numbers of successively installed versions do not have to be contiguous, nor even increasing: e.g., a patched version 0.99.1 could be installed after version 1.0.

Individual library items do not have separate version numbers. Date annotations can be used to indicate which items have changed between two versions of a library.

Libraries can refer to specific versions of other libraries.

library Basic/StructuredDatatypes **version** 1.0
...
from Basic/Numbers **version** 1.0 **get** Nat, Int
...
spec List [**sort** *Elem*] **given** Nat = ...
...
spec Array ... **given** Int = ...
...

Downloading items from particular versions of libraries is necessary if one wants to ensure coherence between libraries. For example, as illustrated above, version 1.0 of Basic/StructuredDatatypes downloads Nat and Int from version 1.0 of Basic/Numbers. Omitting the version number when downloading gives implicitly the *current version* of the library, which may of course change. Even though the developers of libraries may try to ensure backwards compatibility between versions, it could happen that symbols introduced in a new version of a downloaded specification clash with symbols already in use in the library that specified the downloading, causing ill-formedness or inconsistency. So for safety, it is advisable to give explicit version numbers when downloading (also when downloading from version '0' of another library). If one subsequently wants to use symbols that are introduced only in some later version of another library, all that is needed is to change the version number in the downloading(s).

An alternative strategy is to ensure consistency with the *current* versions of all libraries from which specifications are downloaded, by observing the changes in the new versions and adapting the downloading library accordingly. For instance, one might download Int from the current version of Basic/Numbers, instead of from version 1.0 of that library. This may involve extra work when a new version of Basic/Numbers appears, but it has several advantages over the more cautious approach. CASL leaves the choice to the user, although registered libraries will generally be required to use explicit version numbers when downloading from other libraries.

By the way, the current version of a library is not necessarily the one most recently installed: it is the one with the largest version number. As previously mentioned, a patched version 0.99.1 could be installed after version 1.0, but a downloading without an explicit version number would still refer to version 1.0.

Previous versions of libraries are to remain accessible *for ever*... at least in principle. In practice, however, older versions of libraries can be made inaccessible after checking that no (accessible) versions of other libraries download anything from them.

10

Tools

Till Mossakowski

10.1 The Heterogeneous Tool Set (Hets)

The Heterogeneous Tool Set (Hets) is the central analysis tool for CASL

Hets is a tool set for the analysis and encoding of specifications written in CASL, its extensions and sublanguages — hence the name "heterogeneous". The latest version can be obtained under `http://www.tzi.de/cofi`. The installation is easy just follow the instructions.[1]

Consider the first example in this User Manual:

spec STRICT_PARTIAL_ORDER =
%% Let's start with a simple example !
sort *Elem*
pred $\_\_ < \_\_ : Elem \times Elem$ **%%** **pred** abbreviates predicate
$\forall x, y, z : Elem$
- $\neg(x < x)$ %(strict)%
- $x < y \Rightarrow \neg(y < x)$ %(asymmetric)%
- $x < y \wedge y < z \Rightarrow x < z$ %(transitive)%

%{ Note that there may exist x, y such that neither $x < y$ nor $y < x$. **}%**
end

[1] Hets also is available on the CD coming with the CASL reference manual.

Hets can be used for checking static well-formedness of specifications

Let us assume that the example is written to a file named `Order.casl` (actually, this file is provided on the web and on the CD). Then you can check the well-formedness of the specification by typing in

```
hets Order.casl
```

Hets both checks the correctness with respect to the CASL syntax, as well as the static semantics correctness (e.g. whether all identifiers have been declared before they are used, whether the sorting is correct, whether the use of overloaded symbols is unambiguous, and so on).

Now the natural numbers form a strict partial order, which can be expressed by a view as follows:

spec NATURAL = **free type** $Nat ::= 0 \mid suc(Nat)$ **end**

spec NATURAL_ORDER_II =
NATURAL
then **pred** $\_\_ < \_\_ : Nat \times Nat$
$\forall x, y : Nat$
- $0 < suc(x)$
- $\neg x < 0$
- $suc(x) < suc(y) \Leftrightarrow x < y$

end

view V1 : STRICT_PARTIAL_ORDER **to** NATURAL_ORDER_II =
$Elem \mapsto Nat$

end

Again, these specifications can be checked with Hets. However, this checking does only encompass syntactic and static semantic well-formedness — it is *not* checked whether the naturals actually are a strict partial order. Checking this requires theorem proving, which will be covered later.

Hets also displays and manages proof obligations, using development graphs

However, even without theorem proving, it is interesting to inspect the proof obligations arising from a specification. This can be done with

```
hets -g Order.casl
```

Hets now displays a so-called development graph (which is just an overview graph showing the overall structure of the specifications in the library), see Fig. 10.1.

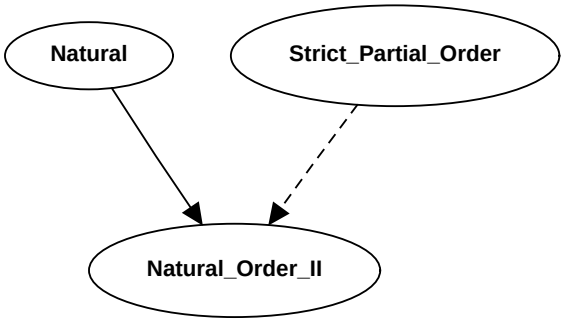

Fig. 10.1. Sample development graph.

Nodes in a development graph correspond to CASL *specifications. Arrows show how basic specification are linked through the structuring constructs*

The solid arrow denotes an ordinary import of specifications (caused by the **then**), while the dashed arrow denotes a proof obligation (caused by the **view**). This proof obligation needs to be discharged in order to show that the view is well-formed.

As a more complex example, consider the following loose specification of an ordering function, taken from Chapter 6:

spec LIST_ORDER_SORTED
[TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then local **pred** $\_\_is\_sorted : List$
$\forall e, e' : Elem;\ L : List$
- $empty\ is\_sorted$
- $cons(e, empty)\ is\_sorted$
- $cons(e, cons(e', L))\ is\_sorted \Leftrightarrow$
  $(cons(e', L)\ is\_sorted \wedge \neg(e' < e))$

within **op** $order : List \to List$
$\forall L : List \bullet order(L)\ is\_sorted$
end

The following specification of insert sort also is taken from Chapter 6:

spec LIST_ORDER [TOTAL_ORDER **with sort** *Elem*, **pred** $\_\_ < \_\_$] =
LIST_SELECTORS [**sort** *Elem*]
then **local** **op** $insert : Elem \times List \to List$
$\forall e, e' : Elem;\ L : List$
- $insert(e, empty) = cons(e, empty)$
- $insert(e, cons(e', L)) = cons(e', insert(e, L))$ *when* $e' < e$ *else* $cons(e, cons(e', L))$

within **op** $order : List \to List$
$\forall e : Elem;\ L : List$
- $order(empty) = empty$
- $order(cons(e, L)) = insert(e, order(L))$

end

Both specifications are related. To see this, we first inspect their signatures. This is possible with

```
hets -g Sorting.casl
```

Hets now displays a more complex development graph, see Fig. 10.2.

Here, we have two types of nodes. The named ones correspond to named specifications, but there are also unnamed nodes corresponding to anonymous basic specifications like the above declaration of the *insert* operation above.

Again, the simple solid arrows denote an ordinary import of specifications (caused by the **then** and **and** keywords in the specifications), while the double arrows denote hiding (caused by the **local**...**then** ... parts in the specifications).

By klicking on the nodes, one can inspect their signatures. In this way, we can see that both LIST_ORDER_SORTED and LIST_ORDER have the same signature. Hence, it is legal to write a view:

view V2 [TOTAL_ORDER] : LIST_ORDER_SORTED[TOTAL_ORDER] **to** LIST_ORDER[TOTAL_ORDER]

end

By again typing

```
hets -g Sorting.casl
```

we now get the following development graph shown in Fig. 10.3.

Internal nodes in a development graph correspond to unnamed parts of a structured specification

In comparison with Fig. 10.2, there are now two new internal nodes, corresponding to the instantiations of both LIST_ORDER_SORTED and LIST_ORDER with TOTAL_ORDER. For each of the instantiations, a proof obligation is generated, which here is a dashed arrow from TOTAL_ORDER to itself, since TOTAL_ORDER is simultaneously the formal and the actual parameter.

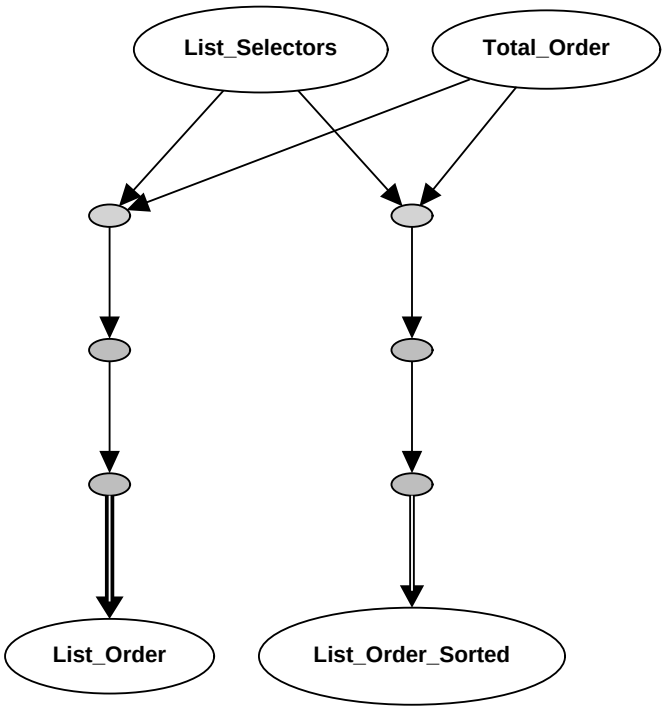

Fig. 10.2. Development graph for the two sorting specifications.

Proof obligations can be discharged in various ways

A trivial proof obligation as the above one can be discharged by Hets alone using the ??? menu. The interesting proof obligation in Fig. 10.3 is the lower dashed arrow between the new nodes. It states that insert sort acutally has the properties of a sorting algorithm. It can be tackeled with the ??? menu. Here, one can choose a theorem prover that shall be used the proof obligation, or just state that one conjectures the obligation to be true.

The menu structure of Hets will be clear in September, hopefully before.

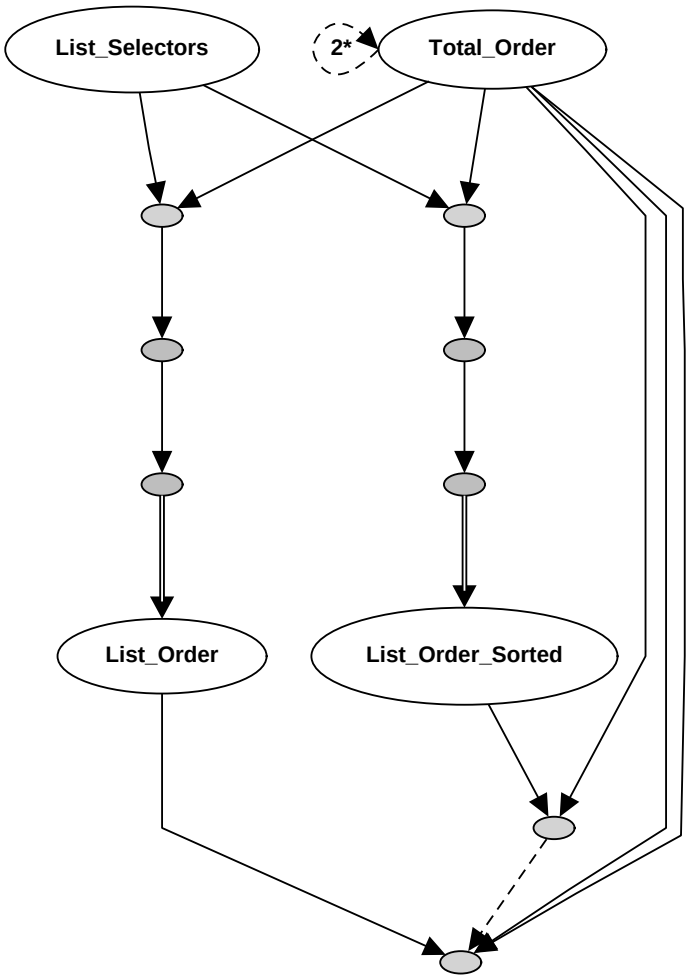

Fig. 10.3. Development graph for the two sorting specifications, with view

10.2 The CASL Tool Set and the Development Graph Manager MAYA

Cats is needed as long as Hets does not support architectural specifications

The CASL Tool Set (Cats) [Mos00] is a precursor of Hets. It comes with roughly the same analysis tools as Hets, but does neither support extensions nor identify sublanguages of CASL. The management of development graphs is not integrated in this tool, but is provided with a different tool called MAYA [AM02] (which only supports part of CASL's structuring constructs).

Cats can be obtained under `http://www.tzi.de/cofi`, while MAYA is available under `http://www.dfki.de/~inka/maya.html`.

Cats and MAYA are mainly important because currently, Hets does not support CASL architectural specifications yet. However, support of architectural specifications is expected to be available within Hets in the near future.

10.3 HOL-CASL

HOL-CASL *is an interactive theorem prover for* CASL*, based on the tactic prover Isabelle*

The HOL-CASL system [Mos00] provides an interface between CASL and the theorem proving system Isabelle/HOL [Pau94]. We have chosen Isabelle because it has a very small core guaranteeing correctness, and its provers like the simplifier or the tableaux prover are built on top of this core. Furthermore, there is over ten years of experience with it (several mathematical textbooks have partially been verified with Isabelle).

Isabelle is a tactic based theorem prover implemented in standard ML. The main Isabelle logic (called Pure) is some weak intuitionistic type theory with polymorphism. The logic Pure is used to represent a variety of logics within Isabelle; one of them being HOL (higher-order logic). For example, logical implication in Pure (written `==>`, also called meta-implication), is different from logical implication in HOL (written `-->`, also called object implication).

CASL now is in turn represented in Isabelle/HOL. Since subsorting and partiality are present in CASL but not in Isabelle/HOL, we have to encode them out, follwoing the lines of [Mos02]. This means that theorem proving is not done in the CASL logic directly, but in the logic HOL (for higher-order logic) of Isabelle. HOL-CASL tries to make user's life easy by

- choosing a shallow embedding of CASL into HOL, e.g. CASL's logical implication `=>` is mapped directly to Isabelle/HOL's logical implication `-->` (and the same holds for other logical connectives and quantifiers), and
- adapting Isabelle/HOL's syntax to be conform with the CASL syntax, e.g. Isabelle/HOL's `-->` is displayed as `=>`, as in CASL.

However, it is essential to be aware of the fact that the Isabelle/HOL logic is different from the CASL logic. Therefore, the formulas appearing in subgoals of proofs with HOL-CASL will not fully conform to the CASL syntax: they may use features of Isabelle/HOL such as higher-order functions that are not present in CASL, and moreover, they also may contain syntax from the meta-logic Pure (e.g. meta-implication `==>`, meta universal quantification `!!` and meta-variables `?x`).

HOL-CASL can be obtained under `http://www.tzi.de/cofi`.

To start the HOL-CASL system, follow the installation instructions, and then type

```
HOL-CASL
```

You can load the above specification file `Order.casl` by typing

```
use_casl "Order";
```

Lets try to prove part of the view V1 above. To prepare proving in the target specification of the view, NATURAL_ORDER_II, type in:

```
CASL_context Natural_Order_II.casl;

AddsimpAll();
```

The first command just selects the specification as current proving context; the second one adds all the axioms of the specification to Isabelle's simplifier (a rewriting engine). Note that here we rely on the axioms being terminating.

To prove the first property expressed by the view, we first have to type in the goal. Then we chose to induct over the variable x, and the rest can be done with automatic simplification. Finally, we name the theorem for later reference:

```
Goal "forall x:Nat . not x<x";
by (induct_tac "x" 1);
by Auto_tac;
qed "Nat_irreflexive";
```

Both Hets and MAYA also provide an interface to HOL-CASL, such that proof obligations arising in development graphs can be discharged using HOL-CASL.

Hets will provide this in the future, hopefully in September — if so, some more words should be added here, otherwise, Hets has to be removed here.

10.4 ELAN-CASL

ELAN-CASL is a rewriting engine for the Horn$^=$ sublanguage CASL

Is this correct? Or is it FOL$^=$ without quantifiers?

ELAN-CASL [KR00] is based on a translation of the Horn$^=$ sublanguage CASL to the input language of the rewriting engine ELAN. It can be obtained under `http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/`.

The required inputs for ELAN are:

- A rewrite program including a signature, a set of rules and a set of strategies.
- A *query* term expressed in the signature of the rewrite program.

The ELAN outputs are the normal forms of the query term, with respect to the rewrite program. Note that because the set of rules is not required to be terminating nor confluent, a query term may have several normal forms, or may not terminate.

ELAN *can be called either as interpreter or as complier.*

Here is an example of how the ELAN interpreter is called:

```
> fenv2elan GT_pred.fenv -sort=bool -interp
To Efix ...
To REF ...
warning: the query sort is bool, index= 428

        ******** ELAN version 3.4 (03/12/99.19:39) ********

         (c)   LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1994 - 1999

Import form file GT_pred

Importing Identifiers
Importing Sorts
Importing Modules
Importing RuleNames
Importing StrategyNames

enter query term finished by the key word 'end':
s(s(zero)) > s(zero) end

[] start with term :
```

```
    s(s(zero))>s(zero)

[] result term:
    true

[] end

enter query term finished by the key word 'end':
s(zero) > s(s(zero)) end

[] start with term :
    s(zero)>s(s(zero))

[] result term:
    false

[] end
```

Here is an example of how the ELAN compiler is called:

```
> fenv2elan GT_pred.fenv -sort=bool -compil -nosplit
To Efix ...
To REF ...
warning: the query sort is bool, index= 428
Reduce ELAN Machine v.2.1
        (c) LORIA (CNRS, INPL, INRIA, UHP, U-Nancy 2), 1998, 1999, 2000
Author: P.-E. Moreau
Reduce ELAN Machine. Reading from file ./GT_pred.ref . . .
Parsing...........***
*** The compiled query is not ground
***
.
Compiling nonamed rules..
Compiling strategies
execute the following line to build your program
        gmake --file ./GT_pred.make
```

Then the generated C code is compiled with the following command:

```
> gmake --file ./GT_pred.make
```

Finally, we get an executable file called `a.out`. This program computes the normal form of any query term:

```
> a.out

Enter a query term of sort 'bool'
```

```
s(s(zero)) > s(zero) end

result = true
```

10.5 ASF+SDF Parser and Structure Editor for CASL

ASF+SDF was used to prototype the CASL *syntax.*

The algebraic specification formalism ASF+SDF [DHK96] and the ASF+SDF Meta-Environment [BDH+01] have been deployed to prototype CASL's concrete syntax, and to develop a mapping for the concrete syntax to an abstract syntax representation as ATerms [BJKO00]. The user-defined (also known as mixfix) syntax of CASL calls for a two-pass approach. In the first pass, the skeleton of a CASL specification is derived, in order to extract user-defined syntax rules. In a second pass, these syntax rules are used to parse the expressions that using the mixfix notation. Currently, only the first pass is realized in SDF, and the parsing is performed based on the underlying Scannerless Generalized LR parsing technology [BSVV02]. A prototype of the mapping from the concrete to abstract representation is written in ASF rewrite rules.

The ASF+SDF Meta-Environment provides syntax-directed editing of CASL *specifications.*

Given the concrete syntax definition of CASL in SDF, syntax-directed editors within the ASF+SDF Meta-Environment come for free. Recent developments in the Meta-Environment [BMV03] allow even the development of a CASL programming environment. Such an environment would include syntax directed editors for CASL and an open archictecture based on a software coordination architecture (ToolBus) [BK98] in order to communicate with other CASL tools, such as the ELAN-CASL toolset.

10.6 Other tools

The following tools could be added, provided until early September there is a stable version available:

- The CASL consistency checker. With the CCC, one can interactively check whether a CASL specification is consistent.
- Translation CASL to OCAML developed at Paris/Poitiers.
- Translation CASL to Haskell developed in Bremen.

11

Foundations

Donald Sannella and Andrzej Tarlecki

A complete presentation of CASL *is in the* CASL Reference Manual.

This CASL *User Manual* has introduced the potential user to the features of CASL mainly by means of illustrative examples. It has presented and discussed the typical ways in which the language concepts and constructs are expected to be used in the course of building system specifications. Thus, the presentation in Chaps. 1–9 focused on *what* the constructs and concepts of CASL are *for*, and *how* they should (and should not) be used. We tried to make these points as clear as possible by referring to simple examples, and by discussing both the general ideas and some details of CASL specifications. We hope that this has given the reader a sufficient feel of the formalism, and enough understanding, to look through the presentation of a basic library of CASL specifications in App. B, to follow the case study in App. C, and to start experimenting with the use of CASL for writing specifications – perhaps employing the support tools presented in Chap. 10.

By no means, however, should this User Manual be regarded as a complete presentation of the CASL specification formalism – this is given in the accompanying volume, the CASL *Reference Manual* [Mos03].

CASL *has a definitive summary.*

Part I of the CASL Reference Manual offers a definitive *summary* of the entire CASL language: all the language constructs are listed there systematically, together with the syntax used to write them down and a detailed explanation of their intended meaning. However, although it tries to be precise and complete, the CASL Summary still relies on natural language to present CASL.

This inherently leaves some room for interpretation and ambiguity in various corners of the language, for example where details of different constructs interact.

CASL *has a complete formal definition.*

A key aim of the entire CoFI initiative is to avoid any potential ambiguities by providing a complete formal definition for CASL, and sound mathematical foundations for the advocated methodology of its use in software specification and development.

Abstract and concrete syntax of CASL *are defined formally.*

We begin by giving a formal definition of the syntax of CASL in Part II of the Reference Manual. *Abstract syntax* is given, where each phrase is written in a way that directly indicates its components, thus making evident its internal structure. In essence, the use of each construct of the language is explicitly labeled here. This is convenient for formal manipulation and analysis, but is not so readable. Therefore, the so-called *concrete syntax* of CASL (as used for instance in the examples throughout this User Manual) is given as well, retaining a direct correspondence with the abstract syntax. This offers to the user of CASL a convenient and readable way of writing down CASL specifications, in a way that makes clear the formal structure of the phrases and constructs used to build them. As usual, the syntax is given as a context-free grammar, using a variant of the BNF notation, relying on well-established theory to give its formal meaning, and on a variety of tools and techniques available for syntactic analysis of languages presented in such a style.

CASL *has a complete formal semantics.*

The ultimate definition of the meaning of CASL specifications is provided by the *semantics* of CASL in Part III of the Reference Manual. The semantics first defines mathematical entities that formally model the intended meaning of various concepts underlying CASL, as already hinted at in Chap. 2, and further introduced and discussed throughout the summary (Part I).

The key concepts here are that of CASL signature, model and formula, together with the satisfaction relation between models and formulae over a common signature; see Chaps. III.2–III.3. In fact, these are variants of the standard algebraic and logical notions, thus linking work on CASL to well-established mathematical theories and ideas.

In a more or less standard way, we use these concepts to build the *semantic domains* which the meanings of phrases in various syntactic categories of CASL inhabit. We have chosen to give the semantics in the form of so-called *natural semantics*, with formal deduction rules to derive judgments concerning the meaning of each CASL phrase from the meanings of its constituent parts. Not only is this a mathematically well-established formalism with an unambiguous interpretation, but we also hope that this makes the semantics itself more readable, with details easier to follow than in some other approaches.

The overall semantics of CASL consists of two parts. The *static semantics* captures a form of static analysis of CASL specifications, in which they are checked for well-formedness – for example, that axioms are well-typed, and references to named entities are in scope. Then the *model semantics* takes a well-formed CASL specification and assigns a model-theoretic meaning to it.

CASL *specifications denote classes of models.*

In CASL, well-formed specifications denote signatures (static semantics) and classes of models (model semantics). Basic specifications, which in essence present a signature and a set of axioms over this signature, denote the class of models that satisfy all the axioms. The semantics of basic specifications is split into two parts: first many-sorted basic specifications are described (Chap. III.2) and then features for defining and using subsorts are added (Chap. III.3).

The semantics is largely institution-independent.

A few more concepts are needed to explicate the semantics of structured specifications (Chap. III.4). Key here is the notion of signature morphism, and the model reducts and translation of sentences that signature morphisms induce. Having introduced those, we obtain the *institution* [GB92] of CASL. One important point of the semantics is that all the layers of the semantics "above" basic specifications are *institution-independent*, i.e., well-defined for any institution chosen to build basic specifications (as long as the institution comes with a bit of extra structure concerned with forming unions of signatures and defining signature morphisms – see Sect. III.4.1).

Next, we have the semantics of architectural specifications (Chap. III.5), which relies on a formal counterpart of the concept of a unit (module) of a system to be developed: self-contained units are viewed simply as models of the underlying institution, and parametrized units as functions mapping such parameter models to result models. Architectural specifications provide a way of specifying the component units of a system and indicating how the overall

system is built by putting these units together. This intuition is captured by the semantics of architectural specifications, which denote a class of permitted unit bindings and a function that maps each such environment to a result unit.

Finally, the libraries layer of CASL is given a rather standard description with libraries modeled as environments giving names to the entities introduced (specifications, architectural specifications, etc.), see Chap. III.6.

The semantics is the ultimate reference for the meanings of all CASL *constructs.*

Overall, the formal mathematical semantics is crucial in the understanding of CASL specifications. It provides their unambiguous meaning, and thus gives the ultimate reference point for all questions concerning the interpretation of any CASL phrase in any context.

We have already experienced how important such a formal semantics may be in the design of CASL itself. In many cases, the intended semantics was prominent in internal discussions on the details of the constructs under consideration, and provided guidelines for many choices in the design of CASL. Indeed, the concrete syntax of CASL was designed only after the semantics was settled.

Proof systems for various layers of CASL *are provided.*

The semantics is also a necessary prerequisite for the development of mechanisms for formal reasoning about CASL specifications. This is dealt with in Part IV of the Reference Manual, where *proof calculi* that support reasoning about the various layers of CASL are presented. The starting point is a formal system of deduction rules which determines a proof-theoretic counterpart of the consequence relation between sets of formulae, thus providing a way for deriving consequences of sets of axioms in CASL specifications. This is then extended to systems of rules for deriving consequences of structured specifications and for proving inclusions between classes of models of such specifications. These systems are also used in rules for formal verification of the internal correctness of system designs as captured by architectural specifications. For all these systems, their soundness is proved and completeness discussed by reference to the formal semantics of CASL.

One point of interest is that, again, the extension of the basic proof system for consequences between sets of formulae to structured and architectural specifications does not really rely on the specifics of the underlying institution, but just reflects the way in which the structuring and architectural constructs are defined for an arbitrary institution.

A formal refinement concept for CASL *specifications is introduced.*

As a basis for the CASL methodology of systematic development of software systems, in Part V of the Reference Manual we provide a formal concept of a correct *refinement step*, leading from one CASL specification to another, presumably more detailed and closer to the final realization of the system under development. The proof systems mentioned above may be applied here for proving correctness of such development steps.

The foundations of our CASL *are rock-solid!*

All this work on the mathematical underpinnings of CASL and related specification and development methodology, as documented in the Reference Manual, should make it exceptionally trustworthy – at least in the sense that it provides a formal point of reference against which all the claims may (and should) be checked.

Appendices

A

CASL Quick Reference

CASL consists of several major *parts*, which are quite independent and may be understood (and used) separately:

Basic specifications

- Denote classes of partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed.
- Subsorts are interpreted as embeddings.
- Axioms are first-order formulae built from definedness assertions and both strong and existential equations. Sort generation constraints can be stated.
- Datatype declarations are provided for concise specification of sorts equipped with some constructors and (optional) selectors, including enumerations and products.

Structured specifications

- Allow translation, reduction, union, and extension of specifications.
- Extensions may be required to be conservative and/or free; initiality constraints are a special case.
- A simple form of generic (parameterized) specifications is provided, together with instantiation involving parameter-fitting translations.

Architectural specifications

- Express that the specified software is to be composed from separately-developed, reusable units with clear interfaces.

Libraries

- Allow the distributed storage and retrieval of (particular versions of) named specifications.

A.1 Basic Specifications

...	%% in any order (but declaration before use):
sorts ...	%% sort declarations and definitions
ops ...	%% operation declarations and definitions
preds ...	%% predicate declarations and definitions
types ...	%% datatype declarations and definitions
generated { ... **}**	%% sort generation constraint
vars ...	%% global variable declarations
$\forall \ldots$	%% universal quantification on lists of axioms
• *F1*	%% lists of axioms
...	
• *Fn*	

A.1.1 Declarations and Definitions

A.1.1.1 Sort Declarations and Definitions

sort s	%% sort declaration
sorts $s1, \ldots, sn$	%% sorts declaration
sorts $s < s'$	%% subsort declaration
sorts $s1, \ldots, sn < s'$	%% subsorts and supersort declaration
sorts $s < s1; \ \ldots; \ s < sn$	%% subsort and supersorts declaration
sorts $s1 = \ldots = sn$	%% isomorphic sorts declaration
sort $s = \{v : s' \bullet F\}$	%% subsort definition

A.1.1.2 Function Declarations and Definitions

op $f : s1 \times \ldots \times sn \to s$	%% total function declaration
op $f : s1 \times \ldots \times sn \to ? \ s$	%% partial function declaration
op $f : s \times s \to s, assoc$	%% associative binary function
op $f : s \times s \to s', comm$	%% commutative binary function
op $f : s \times s \to s, idem$	%% idempotent binary function
op $f : s \times s \to s, unit\ T$	%% unit term for binary function
op $f : s \times s \to s, \ldots, \ldots$	%% multiple function attributes
ops $f1, \ldots, fn : \ldots$	%% functions declaration
op $f(v1 : s1; \ \ldots; \ vn : sn) : s = T$	%% total function definition
op $f(v1 : s1; \ \ldots; \ vn : sn) :? s = T$	%% partial function definition
op $f(\ldots vi1, \ldots, vim : si \ldots) \ldots$	%% abbreviated arguments
ops ...; ...	%% multiple declarations/definitions

A.1.1.3 Constant Declarations and Definitions

op $c : s$ %% constant declaration
op $c :? s$ %% partial constant declaration
op $c1, \ldots, cn : s$ %% constants declaration
op $c : s = T$ %% constant definition
op $c :? s = T$ %% partial constant definition
ops $\ldots ; \ldots$ %% multiple declarations/definitions

A.1.1.4 Predicate Declarations and Definitions

pred $p : s1 \times \ldots \times sn$ %% predicate declaration
pred $p : ( )$ %% constant predicate declaration
preds $p1, \ldots, pn : \ldots$ %% predicates declaration
pred $p(v1 : s1; \ \ldots; \ vn : sn) \Leftrightarrow F$ %% predicate definition
pred $\Leftrightarrow F$ %% constant predicate definition
pred $p(\ldots vi1, \ldots, vim : si \ldots) \ldots$ %% abbreviated arguments
preds $\ldots ; \ldots$ %% multiple declarations/definitions

A.1.1.5 Datatype Declarations

type $s ::= A$ %% datatype declaration with alternatives
types $s1 ::= A1;$ %% multi-sorted datatype declaration
$\ldots ;$
$sn ::= An$
generated types ... %% generated datatype declaration
free types ... %% free datatype declaration

Alternatives (A)

$f(s1'; \ \ldots; \ sk')$ %% total constructor function
$f(s1'; \ \ldots; \ sk')?$ %% partial constructor function
$f(\ldots fi : si \ldots)$ %% total constructor and selector functions
$f(\ldots fi :? si \ldots)$ %% total constructor, partial selector functions
$f(\ldots fi1, \ldots, fim : si \ldots)$ %% abbreviated selectors
c %% constant constructor value
sort s %% subsort
sorts $s1', \ldots, sk'$ %% subsorts
$A1 \mid \ldots \mid Am$ %% multiple alternatives

A.1.1.6 Sort Generation Constraint

generated { **sorts** ... %% generated sorts
ops ... %% generating operations
preds ...
types ... %% generated sorts and generating constructors
}

A.1.2 Variables and Axioms

var $v : s$	%% global variable declaration
vars $v1 : s1; \ldots; vn : sn$	%% global variables declaration
vars $\ldots v1, \ldots, vn : sn \ldots$	%% abbreviated variables declaration
vars $\ldots; \ldots$	%% multiple global variable declarations
$\forall v : s \bullet F1 \ldots \bullet Fn$	%% universal quantification on axiom lists
$\forall v1 : s1; \ldots; vn : sn \bullet \ldots$	%% universal quantifications on axiom lists
$\forall \ldots v1, \ldots, vn : sn \ldots \bullet \ldots$	%% abbreviated quantifications
$\forall \ldots; \ldots \bullet \ldots$	%% multiple quantifications on axiom lists
$\bullet$ $F1$ $\ldots$ $\bullet$ Fn	%% unquantified axiom lists

A.1.2.1 Formulae *F*

$\forall \ldots \bullet F$	%% universal quantification on formula
$\exists \ldots \bullet F$	%% existential quantification
$\exists! \ldots \bullet F$	%% unique-existential quantification
$F1 \wedge \ldots \wedge Fn$	%% conjunction
$F1 \vee \ldots \vee Fn$	%% disjunction
$F \Rightarrow F'$	%% implication
F' *if* F	%% reverse implication
$F \Leftrightarrow F'$	%% equivalence
$\neg F$	%% negation
true	%% truth
false	%% falsity
$p(T1, \ldots, Tn)$	%% predicate application
$t0\ T1\ t1 \ldots Tn\ tn$	%% mixfix predicate application
q	%% constant predicate
$T = T'$	%% ordinary (strong) equality
$T \stackrel{e}{=} T'$	%% existential equality
$T \in s$	%% subsort membership

A.1.2.2 Terms *T*

$f(T1, \ldots, Tn)$	%% application
$t0\ T1\ t1 \ldots Tn\ tn$	%% mixfix application
$t0\ T1, \ldots, Tn\ t1$	%% literal syntax
c	%% constant
$T : s$	%% sorted term
T *as* s	%% projection to subsort
T *when* F *else* T'	%% conditional choice

A.1.3 Symbols

Character set: ASCII (with optional use of ISO Latin-1).

Key Words and Signs

Reserved key words (always lowercase):

and arch as assoc axiom axioms closed comm def else end exists false fit forall free from generated get given hide idem if in lambda library local not op ops pred preds result reveal sort sorts spec then to true type types unit units var vars version view when with within

Reserved key signs:

: :? ::= = => <=> ¬ . · | |−> \/ /\

Unreserved key signs:

< ∗ × −> ? !

Key words representing mathematical signs:

forall exists not in lambda
∀ ∃ ¬ ∈ λ

Key signs representing mathematical signs:

−> => <=> . · |−> /\ \/
→ ⇒ ⇔ • • ↦ ∧ ∨

Identifiers

Sorts and variables are simple words (with digits, primes, and single underscores):

x Y1 Z2′ A_Rather_Long_Identifier .select_1

Operations and predicates can also be sequences of signs (unreserved, and with any brackets [ ]{} balanced):

+−∗/\&=<>[ ]{}!?:.$@#ˆ˜¡¿×÷£©±¶§¹²³·∉°¬µ|

or single decimal digits *1234567890*, or single quoted characters *′c′*.

The signs ( )[ ];,‘”% are *not* allowed in identifiers, nor are the ISO Latin-1 signs for general currency, yen, broken vertical bar, registered trade mark, masculine and feminine ordinals, left and right angle quotes, fractions, soft hyphen, acute accent, cedilla, macron, and umlaut.

Function and predicate identifiers can also be infixes, prefixes, postfixes, and general mixfixes, formed from words and/or sequences of signs separated by *double* underscores (indicating the positions of the arguments), with any brackets [ ]{} balanced:

$\_\_{+}{+}\_\_$ $||\_\_||$ $\{[\_\_]\}$ $Push\_\_onto\_\_$

Invisible mixfix identifiers (such as $\_\_\_\_$) with two or more arguments are allowed. (Subsort embeddings give the effect of invisible unary functions.)

A sort, operation, or predicate identifier can be compound, with a list of identifiers appended to its final token.

Literal Strings and Numbers

"*this is a string*" *42* *3.14159* $1E-9$ *27.3e6*

Library Identifiers

Names of libraries are either paths, e.g.:

BASIC/NUMBERS BASIC/ALGEBRA_II

or URLs formed from A...ZA...z0...9$-_@.&+!*"'(),~ and hexadecimal codes %XX, and prefixed by HTTP://, FTP://, or FILE:///.

Version numbers of libraries are hierarchical: 0, 0.999, 1.0.2.

A.1.4 Comments

%% This is a comment at the end of a line...
...**%{** This is an in-line comment **}%** ...
...**%{** This a comment that might take
several lines **}%**
%[This is for commenting-out text
%{ including other kinds of comment **}%** **]%**

A.1.5 Annotations

A label is of the form %(text)%.

An end-of-line annotation is of the general form **%word_s** ...
with a space following the word_s.

A possibly multi-line annotation is of the general form **%word(** ... **)%**
with *no* space preceding the '('.

A.2 Structured Specifications

A.2.1 Specifications (*SP*)

SP **with** *SM* %% symbol translation
SP **hide** *SL* %% hiding listed symbols
SP **reveal** *SM* %% revealing/translating symbols
SP1 **and** ... **and** *SPn* %% union
SP1 **then** ... **then** *SPn* %% extension
free *SP* %% free or initial
local *SP* **within** *SP′* %% implicit hiding of local symbols
closed *SP* %% non-extension
SN %% reference to named specification
SN[*FA1*]...[*FAn*] %% instantiation of generic specification

A.2.2 Named and Generic Specifications

spec *SN* = *SP* **end** %% named specification (**end** optional)
spec *SN*[*SP1*]...[*SPn*] = *SP* %% generic specification
spec *SN*[*SP1*]...[*SPn*] %% generic with imports
given *SP1″*,...,*SPm″* =
SP

A.2.2.1 Fitting Arguments (*FA*)

SP′ **fit** *SM* %% fitting by symbol map
SP′ %% implicit fitting
FV %% fitting view

A.2.3 Named and Parametrized Views

view *VN*: *SP* **to** *SP′* = %% named view (**end** optional)
SM **end**
view *VN*[*SP1*]...[*SPn*] %% generic view
: *SP* **to** *SP′* =
SM
view *VN*[*SP1*]...[*SPn*] %% generic view with imports
given *SP1″*,...,*SPm″*
: *SP* **to** *SP′* =
SM

A.2.3.1 Fitting Views (*FV*)

view *VN* %% reference to named view
view *VN*[*FA1*]...[*FAn*] %% instantiation of generic view

A.2.4 Symbol Lists (*SL*) and Maps (*SM*)

$SY1, \ldots, SYn$	%% lists (maybe with *sorts*, *ops*, *preds*)
$SY1 \mapsto SY1', \ldots, SYn \mapsto SYn'$	%% maps (maybe with *sorts*, *ops*, *preds*)
$\ldots, SYi, \ldots$	%% abbreviates $\ldots, SYi \mapsto SYi, \ldots$

A.3 Architectural Specifications

A.3.1 Named Specifications

arch spec $ASN = ASP$ **end**	%% named arch. spec.
unit spec $SN = USP$ **end**	%% named unit spec.

A.3.2 Architectural Specifications (*ASP*)

ASN	%% arch. spec. name
units $UD1; \ldots; UDn$ **result** UE	%% basic arch. spec.

A.3.3 Unit Specifications (*USP*)

SP	%% unit type
$SP1 \times \ldots \times SPn$ **to** SP	%% functional unit type
closed USP	%% non-extension
arch spec ASP	%% models of arch. spec.

A.3.4 Unit Declarations and Definitions (*UD*)

$UN : USP$	%% unit declaration
$UN : USP$ **given** $UT1, \ldots, UTn$	%% importing unit declaration
$UN = UE$	%% unit definition

A.3.5 Unit Expressions (*UE*)

UT	%% unit term
$\lambda UN1 : SP1; \ldots; UNn : SPn \bullet UT$	%% unit composition

A.3.6 Unit Terms (*UT*)

UT **with** SM	%% symbol translation
UT **hide** SL	%% hiding listed symbols
UT **reveal** SM	%% revealing/translating symbols
$UT1$ **and** $\ldots$ **and** UTn	%% amalgamation
local $UD1; \ldots; UDn$ **within** UT	%% local units
UN	%% unit name
$UN[UT1$ **fit** $SM1] \ldots [UTn$ **fit** $SMn]$	%% functional unit application

A.4 Libraries

library *LN* ... %% named library of downloadings, specifications, views

A.4.1 Downloadings

from *LN* **get** *IN1*,..., *INn* **end** %% downloads listed items
from *LN* **get** ... *IN* $\mapsto$ *IN′* ... **end** %% renames downloaded items

A.4.2 Library Names (*LN*)

BASIC/NUMBERS %% greatest version installed
BASIC/ALGEBRA_II **version** *0.999* %% specified version installed
HTTP://... %% greatest version uninstalled
HTTP://... **version** *1.0.2* %% specified version uninstalled

B

Basic Library

Markus Roggenbach, Till Mossakowski, and Lutz Schröder

The CASL Basic Library consist of specifications of often-needed datatypes and view among them, freeing the specifier from re-inventing well-know things. This can be compared to standard libraries in programming languages. While this user manual often discusses several styles of specifying with CASL, the basic datatypes consistently follow a certain style described in [Mos03, VI.12].

We here describe two of the libraries (cf. the overview in Fig. B.1): the library of numbers and of structured datatypes. We also provide stripped-down versions of the libraries themselves, with some of the specification and all axioms and annotations removed. This stripped-down version can serve for getting a first overview over the signatures of the specified datatypes.

The full CASL Basic Library is printed in the CASL Reference Manual [Mos03, VI]. The latest version is available at

`http://www.tzi.de/cofi`

Possibly also in the CD coming with this user manual, if there is one?

The Hets tool described in Chap. 10 allows you to obtain a graphical overview over the specifications in the libraries and also inspect their signatures. We recommend to do this in order to get a better overview, and also for answering specific questions that arise when using the basic datatypes.

B.1 Library Basic/Numbers

This library provides monomorphic specifications of natural numbers, integers, rational numbers, and rational numbers.

In the specification NAT, the natural numbers are specified as a free type, together with a bunch of predicates and operations over the sort *Nat* of natural numbers.

Note that the names for the *partial operations* subtraction $\_\_-?\_\_$ and division $\_\_/?\_\_$ include a question mark. This is to avoid overloading with the *total operations* $\_\_-\_\_$ on integers and $\_\_/\_\_$ on rationals, which would lead to

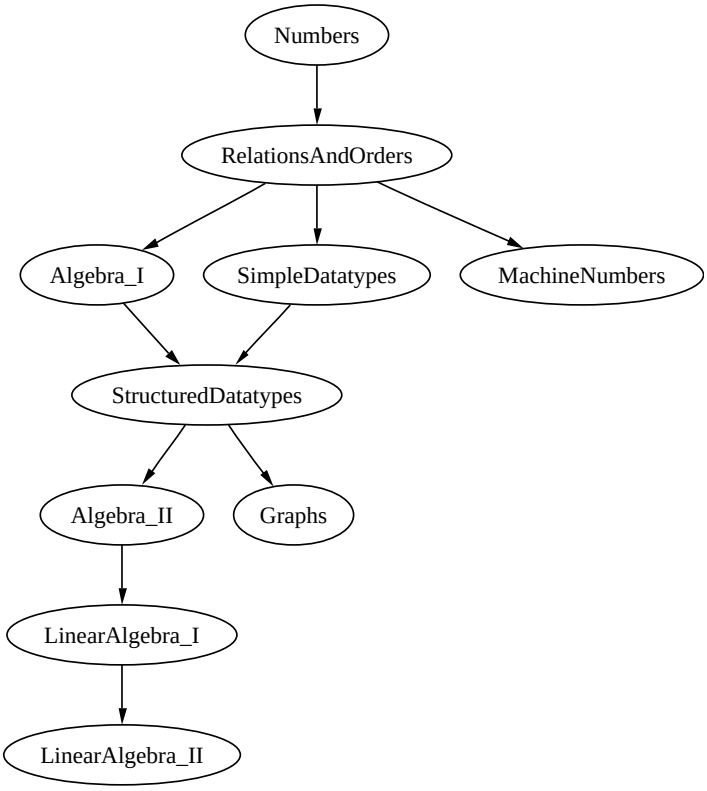

Fig. B.1. Dependency graph of the libraries of basic datatypes.

inconsistencies as both these specifications import the specification of natural numbers.

The introduction of the subsort *Pos*, consisting of the positive naturals, gives rise to certain new operations, e.g.

$\_\_*\_\_ : Pos \times Pos \to Pos$,

whose semantics is completely determined by overloading.

The specification INT of integers is built on top of the specification of naturals: integers are defined as equivalence classes of pairs of naturals written as differences — the axioms (which are omitted in the specification below)

specify that two parts are equivalent if their differences are equal. The sort *Nat* is then declared to be a subsort of *Int*.

Besides the division operator __/?__, the specification INT also provides the function pairs *div*/*mod* and *quot*/*rem*, respectively, as constructs for division — behaving differently on the negative numbers, see [Mos03, VI.2] for a discussion.

The specification RAT of rational numbers follows the same scheme as the specification of integers discussed above. This time, the specification INT is imported. The rationals are then defined as equivalence classes of pairs consisting of an integer and a positive number written as quotients. The sort *Int* is then declared to be a subsort of *Rat*. Note that thanks to CASL subsorting, the declaration of the operation

$\_\_/\_\_ : Rat \times Rat \to ? Rat;$

allows writing rationals also as pairs x/y of arbitrary integers x and $y \neq 0$.

Again, the definition of the predicates and operations on rationals usually covers the whole domain of rationals, i.e. concerning consistency one has to show that they do not contradict to the axioms for naturals and integers, to which they are related by overloading.

library BASIC/NUMBERS

%{ This library provides specifications of naturals, integers, and rationals. Concerning the rationals, the specification Rat includes the datatype proper, while the specification DecimalFraction adds the notions needed to represent rationals as decimal fractions. **}%**

spec NAT =
  free type $Nat ::= 0 \mid suc(pre{:}?Nat)$
  preds $\_\_\leq\_\_, \_\_\geq\_\_, \_\_<\_\_, \_\_>\_\_ : Nat \times Nat;$
    $even, odd : Nat$
  ops $\_\_! : Nat \to Nat;$
    $\_\_+\_\_, \_\_*\_\_, \_\_\hat{}\_\_, min, max, \_\_-!\_\_ : Nat \times Nat \to Nat;$
    $\_\_-?\_\_, \_\_/?\_\_, \_\_div\_\_, \_\_mod\_\_, gcd : Nat \times Nat \to ? Nat$
    %% Operations to represent natural numbers with digits:
  ops $1 : Nat = suc(0);$ %(1_def_Nat)%
    $2 : Nat = suc(1);$ %(2_def_Nat)%
    $3 : Nat = suc(2);$ %(3_def_Nat)%
    $4 : Nat = suc(3);$ %(4_def_Nat)%
    $5 : Nat = suc(4);$ %(5_def_Nat)%
    $6 : Nat = suc(5);$ %(6_def_Nat)%
    $7 : Nat = suc(6);$ %(7_def_Nat)%
    $8 : Nat = suc(7);$ %(8_def_Nat)%
    $9 : Nat = suc(8);$ %(9_def_Nat)%

```
        __@@__(m: Nat; n: Nat): Nat = (m * suc(9)) + n %(decimal_def)%
sort    Pos = {p: Nat • p > 0}
ops     1: Pos = suc(0);                                    %(1_as_Pos_def)%
        __*__ : Pos × Pos→ Pos;
        __+__ : Pos × Nat→ Pos;
        __+__ : Nat × Pos→ Pos;
        suc : Nat→ Pos
end

spec INT =
     NAT
then
     generated type Int ::= __−__(Nat; Nat)
     sort   Nat < Int
            %% a system of representatives for sort Int is
            %% a - 0 and 0 - p, where a: Nat and p: Pos
     preds __≤__, __≥__, __<__, __>__ : Int × Int;
            even, odd : Int
     ops    −__, sign : Int→ Int;
            abs : Int→ Nat;
            __+__, __*__, __−__, min, max : Int × Int→ Int;
            __^__ : Int × Nat→ Int;
            __/?__, __div__, __quot__, __rem__ : Int × Int →? Int;
            __mod__ : Int × Int →? Nat
end

spec RAT =
     INT
then
     generated type Rat ::= __/__(Int; Pos)
     sort   Int < Rat
     preds __≤__, __<__, __≥__, __>__ : Rat × Rat
     ops    −__, abs : Rat→ Rat;
            __+__, __−__, __*__, min, max : Rat × Rat→ Rat;
            __/__ : Rat × Rat →? Rat;
            __^__ : Rat × Int→ Rat
end
```

B.2 Library Basic/StructuredDatatypes

This library provides specifications which formalize structuring concepts of data as used e.g. for the design of algorithms or within programming languages. Its main focus are advanced concepts as e.g. (finite) sets, lists, strings,

(finite) maps, (finite) bags, arrays, and various kinds of trees. But is also covers some elementary constructions like the encapsulation of data within a 'maybe'-type or arranging data as pairs. Common to all these concepts is that they are generic. Consequently, all specifications of this library are parameterized.

Finite sets, finite maps and finite bags are specified in terms of observers: given a generated sort, an operation or predicate is introduced in order to define equality on this sort. Concerning finite sets, equality on the sort *FinSet*[*Elem*] is characterized using the predicate *__eps__* (displayed as ϵ) in the specification GENERATEFINSET. Finite maps, i.e. elements of the sort *FinMap*[S, T], are considered to be identical, if their evaluation under the operation *eval* yields the same result, c.f. the specification GENERATEFINMAP. In the specification GENERATEBAG, those elements of sort *Bag*[*Elem*] are identified that show the same frequency (observed by the operation *freq*) for all entries.

Lists are specified in terms of a free datatype. In the specification GENERATELIST, lists are built up from the empty list by adding elements in front. The usual list operations are provided. *first* and *last* select the first or last element of a list, while *rest* or *front* select the remaining list. *#__* counts the number of elements in a list. *take* takes the first n elements of a list, while *drop* drops them.

The specification ARRAY includes the condition $min <= max$ as an axiom in its first parameter. This ensures a non-empty index set. Arrays are defined as finite maps from the sort *Index* to the sort *Elem*, where the typical array operations evaluation and assignment are introduced in terms of finite map operations. Finally, revealing the essential signature elements yields the desired datatype.

The library concludes with several specifications concerning trees. There are specifications covering binary trees (BINTREE, BINTREE2), k-branching trees (KTREE), and trees with a possibly different branching at each node (NTREE). Each of these branching structures can be equipped with data in different ways: Either all nodes of a tree carry data (as it is the case in BINTREE, KTREE, and NTREE), or just the leaves of a tree have a data entry (as in BINTREE2).

library BASIC/STRUCTUREDDATATYPES

from BASIC/NUMBERS **version** 1.0 **get** NAT, INT

spec MAYBE [**sort** S] = **%mono**
  free type *Maybe*[S] ::= $\bot$ | *just*(*val*:?S)
end

spec PAIR [**sort** S] [**sort** T] = **%mono**
  free type *Pair*[S,T] ::= *pair*(*first*:S; *second*:T)

end
spec GENERATEFINSET [**sort** *Elem*] = **%mono**
generated type *FinSet*[*Elem*] ::= {} | __+__(*FinSet*[*Elem*]; *Elem*)
pred __ϵ__ : *Elem* × *FinSet*[*Elem*]
%% a system of representatives for sort FinSet[Elem] is
%%
%% {} and {} + x_1 + x_2 + ... + x_n
%%
%% where x_1 <= x_2 <= ... <= x_n, n >= 1, x_i of type Elem,
%% for an arbitrary total order __<=__ on Elem.
end

spec FINSET [**sort** *Elem*] **given** NAT = **%mono**
GENERATEFINSET [**sort** *Elem*]
then %def
preds *isNonEmpty* : *FinSet*[*Elem*];
__$\subseteq$__ : *FinSet*[*Elem*] × *FinSet*[*Elem*]
ops {__} : *Elem*→ *FinSet*[*Elem*];
$\sharp$__ : *FinSet*[*Elem*]→ *Nat*;
__+__ : *Elem* × *FinSet*[*Elem*]→ *FinSet*[*Elem*];
__−__ : *FinSet*[*Elem*] × *Elem*→ *FinSet*[*Elem*];
__$\cap$__, __$\cup$__, __−__, __*symDif*__ :
FinSet[*Elem*] × *FinSet*[*Elem*]→ *FinSet*[*Elem*]
end

spec GENERATELIST [**sort** *Elem*] =
free type *List*[*Elem*] ::= [] | __::__(*first*:?*Elem*; *rest*:?*List*[*Elem*])
end

spec LIST [**sort** *Elem*] **given** NAT =
GENERATELIST [**sort** *Elem*]
then
preds *isEmpty* : *List*[*Elem*];
__ϵ__ : *Elem* × *List*[*Elem*]
ops __+__ : *List*[*Elem*] × *Elem*→ *List*[*Elem*];
first, *last* : *List*[*Elem*] →? *Elem*;
front, *rest* : *List*[*Elem*] →? *List*[*Elem*];
$\sharp$__ : *List*[*Elem*]→ *Nat*;
__++__ : *List*[*Elem*] × *List*[*Elem*]→ *List*[*Elem*];
reverse : *List*[*Elem*]→ *List*[*Elem*];
__!__ : *List*[*Elem*] × *Nat* →? *Elem*;
take, *drop* : *Nat* × *List*[*Elem*] →? *List*[*Elem*];
freq : *List*[*Elem*] × *Elem*→ *Nat*
end

spec GENERATEFINMAP [**sort** S] [**sort** T] = **%mono**
generated type
$FinMap[S,T] ::= empty \mid \_\_[\_\_/\_\_](FinMap[S,T];\ T;\ S)$
op $eval : S \times FinMap[S,T] \to ?\ T$
end

spec FINMAP [**sort** S] [**sort** T] **given** NAT = **%mono**
GENERATEFINMAP [**sort** S] [**sort** T]
and
FINSET [**sort** S **fit** $Elem \mapsto S$]
and
FINSET [**sort** T **fit** $Elem \mapsto T$]
then %def
free type $Entry[S,T] ::= [\_\_/\_\_](target{:}T;\ source{:}S)$
preds $isEmpty : FinMap[S,T]$;
$\_\_\epsilon\_\_ : Entry[S,T] \times FinMap[S,T]$;
$\_\_::\_\_- >\_\_ : FinMap[S,T] \times FinSet[S] \times FinSet[T]$
ops $\_\_+\_\_,\ \_\_-\_\_ : FinMap[S,T] \times Entry[S,T] \to FinMap[S,T]$;
$\_\_-\_\_ : FinMap[S,T] \times S \to FinMap[S,T]$;
$\_\_-\_\_ : FinMap[S,T] \times T \to FinMap[S,T]$;
$dom : FinMap[S,T] \to FinSet[S]$;
$range : FinMap[S,T] \to FinSet[T]$;
$\_\_\cup\_\_ : FinMap[S,T] \times FinMap[S,T] \to ?\ FinMap[S,T]$
end

spec GENERATEBAG [**sort** $Elem$] **given** NAT = **%mono**
generated type $Bag[Elem] ::= \{\} \mid \_\_+\_\_(Bag[Elem];\ Elem)$
op $freq : Bag[Elem] \times Elem \to Nat$
end

spec BAG [**sort** $Elem$] **given** NAT = **%mono**
GENERATEBAG [**sort** $Elem$]
then
preds $isEmpty : Bag[Elem]$;
$\_\_\epsilon\_\_ : Elem \times Bag[Elem]$;
$\_\_\subseteq\_\_ : Bag[Elem] \times Bag[Elem]$
ops $\_\_+\_\_ : Elem \times Bag[Elem] \to Bag[Elem]$;
$\_\_-\_\_ : Bag[Elem] \times Elem \to Bag[Elem]$;
$\_\_-\_\_,\ \_\_\cup\_\_,\ \_\_\cap\_\_ : Bag[Elem] \times Bag[Elem] \to Bag[Elem]$
end

spec ARRAY [**ops** $min, max : Int$
$\bullet\ min \leq max$ %(Cond_nonEmptyIndex)%]
[**sort** $Elem$]

```
      given INT =
      sort   Index = {i: Int • min ≤ i ∧ i ≤ max}
then
      {      FINMAP [sort Index fit sort S ↦ Index]
             [sort Elem fit sort T ↦ Elem]
             with sort FinMap[Index,Elem] ↦ Array[Elem], op empty ↦ init
       then
             ops    __!__:=__ : Array[Elem] × Index × Elem→ Array[Elem];
                    __!__ : Array[Elem] × Index →? Elem
      }
      reveal sort Array[Elem], ops init, __!__, __!__:=__
end

spec GENERATEBINTREE [sort Elem] =
      free type
          BinTree[Elem] ::= nil
                          | binTree(entry:?Elem; left:?BinTree[Elem];
                                    right:?BinTree[Elem])
end

spec BINTREE [sort Elem] given NAT =
      GENERATEBINTREE [sort Elem]
and
      FINSET [sort Elem]
then
      preds isEmpty, isLeaf : BinTree[Elem];
            isCompoundTree : BinTree[Elem];
            __∈__ : Elem × BinTree[Elem]
      ops   height : BinTree[Elem]→ Nat;
            leaves : BinTree[Elem]→ FinSet[Elem]
end

spec GENERATEBINTREE2 [sort Elem] = %mono
      free type
          NonEmptyBinTree2[Elem] ::= leaf(entry:?Elem)
                                   | binTree(left:?NonEmptyBinTree2[Elem];
                                             right:?NonEmptyBinTree2[Elem])
      free type BinTree2[Elem] ::= nil | sort NonEmptyBinTree2[Elem]
end

spec BINTREE2 [sort Elem] given NAT = %mono
      GENERATEBINTREE2 [sort Elem]
and
      FINSET [sort Elem]
then %def
```

```
    preds isEmpty, isLeaf : BinTree2[Elem];
          isCompoundTree : BinTree2[Elem];
          __∈__ : Elem × BinTree2[Elem]
    ops   height : BinTree2[Elem]→ Nat;
          leaves : BinTree2[Elem]→ FinSet[Elem]
end

spec GenerateKTree [op k : Int
                    • k ≥ 1                %(Cond_nonEmptyBranching)%]
                    [sort Elem]
    given Int =
    Array [ops 1 : Int;
                                  k : Int
               fit ops min : Int ↦ 1, max : Int ↦ k]
          [sort KTree[k,Elem] fit sort Elem ↦ KTree[k,Elem]]
then
    free type
        KTree[k,Elem] ::= nil
                        | kTree(entry:?Elem; branches:?Array[KTree[k,Elem]])
end

spec KTree [op k : Int
            • k ≥ 1                    %(Cond_nonEmptyBranching)%]
           [sort Elem]
    given Int =
    GenerateKTree [op k : Int] [sort Elem]
and
    FinSet [sort Elem]
then %def
    preds isEmpty, isLeaf : KTree[k,Elem];
          isCompoundTree : KTree[k,Elem];
          __∈__ : Elem × KTree[k,Elem]
    ops   height : KTree[k,Elem]→ Nat;
          maxHeight : Index × Array[KTree[k,Elem]]→ Nat;
          leaves : KTree[k,Elem]→ FinSet[Elem];
          allLeaves : Index × Array[KTree[k,Elem]]→ FinSet[Elem]
end

spec GenerateNTree [sort Elem] =
    List [sort NTree[Elem] fit sort Elem ↦ NTree[Elem]]
then
    free type
        NTree[Elem] ::= nil
                      | nTree(entry:?Elem; branches:?List[NTree[Elem]])
end
```

spec NTree [**sort** *Elem*] **given** Nat =
GenerateNTree [**sort** *Elem*]
and
FinSet [**sort** *Elem*]
then
preds *isEmpty*, *isLeaf* : *NTree*[*Elem*];
isCompoundTree : *NTree*[*Elem*];
$\_\_\epsilon\_\_$: *Elem* × *NTree*[*Elem*]
ops *height* : *NTree*[*Elem*]→ *Nat*;
maxHeight : *List*[*NTree*[*Elem*]]→ *Nat*;
leaves : *NTree*[*Elem*]→ *FinSet*[*Elem*];
allLeaves : *List*[*NTree*[*Elem*]]→ *FinSet*[*Elem*]
end

C

Case Study: The Steam-Boiler Control System

This is a *preliminary draft*, and various parts of it still need to be checked and polished. Suggestions for improvements are most welcome, as well as detailed comments on the specifications.

In this chapter we illustrate the use of CASL on a fairly large and complex case study, the *Steam-boiler control system*. We describe how to derive a CASL specification of the steam-boiler control system, starting from the informal requirements provided to the participants of the Dagstuhl Meeting *Methods for Semantics and Specification*, organized jointly by Jean-Raymond Abrial, Egon Börger and Hans Langmaack in June 1995.[1] The aim of this formalization process is to analyze the informal requirements, to detect inconsistencies and loose ends, and to translate the requirements into a formal, algebraic, CASL specification. During this process we have to provide interpretations for the unclear or missing parts. We explain how we can keep track of these additional interpretations by localizing very precisely in the formal specification where they lead to specific axioms, thereby taking care of the traceability issues. We also explain how the CASL specification is obtained in a stepwise way by successive analysis of various parts of the problem description. Emphasis is put on how to specify the detection of the steam-boiler failures. Finally we discuss validation of the CASL requirements specification resulting from the formalization process, and in a last step we refine the requirements specification in a sequence of architectural specifications that describe the intended architecture of the implementation of the steam-boiler control system.

Somewhere a mention of Bidoit & al. paper in LNCS 1165 should be added.

C.1 Introduction

The aim of this chapter is to illustrate how one can solve the "Steam-boiler control specification problem" (see Sec. C.12) with CASL. For this we provide a CASL specification of the software system that controls the level of water in the steam-boiler. Our work plan can be described as follows:

[1] The steam-boiler control specification problem is reproduced in Sec. C.12.

1. The main task is to derive from the informal requirements a requirements specification, written in CASL, of the steam-boiler control system. In particular, this task involves the following activities:
   a) We must proceed to an in-depth analysis of the informal requirements. Obviously, this is necessary to gain a sufficient understanding of the problem to be specified, and this preliminary task may not seem worth mentioning. Let us stress, however, that the kind of preliminary analysis required for writing a formal specification proves especially useful to detect *discrepancies* in the informal requirements that would otherwise be very difficult to detect. Indeed, from our practical experience, this step is usually very fruitful from an engineering point of view, and one could argue that the benefits to be expected here are enough in themselves to justify the use of formal methods, even if for lack of time (or other resources) no full formal development of the system is performed.
   b) Once we have a sufficient understanding of the problem to be specified, we must translate the informal requirements into a formal specification. This step will require us to provide interpretations for the unclear or missing parts of the informal requirements. Moreover, this formalization process will also be helpful to further detect inconsistencies and loose ends in the informal requirements. Here, a very important issue is to keep track of the interpretations made during the formalization process, in order to be able, later on, to take into account further modifications and changes of the informal requirements.
   c) When we have written the formal requirements specification, we must carefully check its adequacy with respect to the informal requirements: this part is called the validation of the formal specification.

   In principle there should be some interaction between the specification team and the team who has designed the informal requirements, in particular to check whether the suggested interpretations of the detected loose ends are adequate. In the framework of this case study, however, such interactions were not possible, and we can only use our intuition to assess the well-foundedness of the interpretations made during the writing of the formal specification.
2. Once a validated requirements specification is obtained, we can proceed toward a program by a sequence of refinements. Here a crucial step is the choice of an architecture of the desired implementation, expressed by an architectural specification as explained in Chap. 8. Each refinement step leads to proof obligations which allow the correctness of the performed refinement to be assessed. In a last step, a program is derived from the final design specification.

Is a pointer to [Mos03, Part V] needed here?

Before starting to explain how to write a CASL requirements specification of the steam-boiler control system, let us make a few comments on this case study. First, note that, although in principle a *hybrid system*, the steam-

boiler control system turns out to be merely a reactive (not even a "hard real-time") system (see e.g. the assumptions made in Sec. C.12.3). Moreover, even if the whole system (i.e., the control program and its physical environment) is distributed, this is not the case (at least at the requirements level) for the steam-boiler control system. CASL turns out to be especially well-suited to capture the features of systems like the steam-boiler control system, where data and control are equally important (in particular, here data play a prominent role in failure detection). The various constructs provided by CASL allow the specifications to be formulated straightforwardly and perspicuously – and significantly more concisely than in other algebraic specification languages.

As a last remark we must make clear that for the sake of simplicity the *initialization* phase of the steam-boiler control system (see Sec. C.12.4.1) is not specified. However, it should be clear that it would be straightforward to extend our specification so as to take the initialization phase into account, following exactly the same methodology as for the rest of the case study.

This chapter is organized as follows. In Sec. C.2 we start by providing some elementary specifications that will be useful for the rest of the case study. In Sec. C.3 we explain how we will proceed to derive the CASL requirements specification in a stepwise way. Then in Sec. C.4 we detail the specification of the mode of operation of the steam-boiler control system. In Sec. C.5 we specify the detection of the various equipment failures, and in Sec. C.6 we explain how we can compute, at each cycle, some predicted values for the messages to be received at the next cycle. In Sec. C.9 we explain how our CASL requirements specification can be validated, and in Sec. C.10 we refine the CASL requirements specification in a sequence of architectural specifications that describe the intended architecture of the implementation of the steam-boiler control system. Finally in Sec. C.11 we offer some concluding remarks.

C.2 Getting Started

As explained in Sec. C.12.3, in each cycle the steam-boiler control system collects the messages received, performs some analysis of the information contained in them, and then sends messages to the physical units. We will therefore start with the specification of some elementary datatypes, such as "received messages" and "sent messages". To specify the messages sent and received, we follow Secs. C.12.5 and C.12.6. Note that some messages have parameters (e.g. pump number, pump state, pump controller state, mode of operation), and we must therefore specify the corresponding datatypes as well. For the sake of clarity, we group together all similar messages (e.g. all "repaired" messages, all "failure acknowledgement" messages) by introducing a suitable parameter "physical unit". A physical unit is either a pump, a pump controller, the water level measuring device or the output of steam measuring device. Remember that we do not specify the physical units as such, since we do not specify the physical environment of the steam-boiler (we do not spec-

ify the steam-boiler either, we only specify the steam-boiler control system). Hence the datatype "physical unit" is just an elementary datatype that says that we have some pumps, some pump controllers, and the two measuring devices.

Some messages have a value v as parameter. From the informal requirements we can infer that these values are (approximations of) real numbers, but indeed it is not necessary at this level to make any decision about the exact specification of these values. In our case study, we will therefore rely on a very abstract (loose) specification VALUE, introducing a sort *Value* together with some operations and predicates, which are left unspecified (we expect of course that these operations and predicates will have the intuitive interpretation suggested by their names). This means that we consider VALUE as being a general parameter of our specification.[2] This point is discussed again in Sec. C.10. Note also that we will abstract from measuring units (such as litre, litre/sec), since ensuring that these units are consistently used is a very minor aspect of this particular case study.[3]

This first analysis leads to the following specifications: VALUE, BASICS, MESSAGES_SENT, and MESSAGES_RECEIVED.

from BASIC/NUMBERS **get** NAT

spec VALUE =
%% At this level we don't care about the exact specification of values.
NAT
then **sorts** $Nat < Value$
ops $0, 1 : Nat$;
$\_\_ + \_\_ : Value \times Value \to Value, assoc, comm, unit\ 0$;
$\_\_ - \_\_ : Value \times Value \to Value$;
$\_\_ \times \_\_ : Value \times Value \to Value, assoc, comm, unit\ 1$;
$\_\_/2, \_\_^2 : Value \to Value$;
$min, max : Value \times Value \to Value$
preds $\_\_ < \_\_, \_\_ \leq \_\_ : Value \times Value$
end

[2] We leave VALUE as an implicit parameter of our specifications, rather than using generic specifications taking VALUE as a parameter, since our specifications are not to be instantiated by argument specifications describing several kinds of values, but on the contrary should all refer to the same abstract datatype of values.

[3] It is of course possible to take measuring units into account, following for instance the method described in [CRV03]. Appropriate CASL libraries supporting measuring units are currently being developed.

```
spec BASICS =
    free type PumpNumber ::= Pump1 | Pump2 | Pump3 | Pump4;
    free type PumpState ::= Open | Closed;
    free type PumpControllerState ::= Flow | NoFlow;
    free type PhysicalUnit ::= Pump(PumpNumber)
                             | PumpController(PumpNumber)
                             | OutputOfSteam | WaterLevel;
    free type Mode ::= Initialization | Normal | Degraded
                     | Rescue | EmergencyStop;
end

spec MESSAGES_SENT =
    BASICS
then free type
    S_Message ::= MODE(Mode) | PROGRAM_READY | VALVE
                | OPEN_PUMP(PumpNumber)
                | CLOSE_PUMP(PumpNumber)
                | FAILURE_DETECTION(PhysicalUnit)
                | REPAIRED_ACKNOWLEDGEMENT(PhysicalUnit);
end

spec MESSAGES_RECEIVED =
    BASICS and VALUE
then free type
    R_Message ::= STOP | STEAM_BOILER_WAITING
                | PHYSICAL_UNITS_READY
                | PUMP_STATE(PumpNumber; PumpState)
                | PUMP_CONTROLLER_STATE(PumpNumber;
                                        PumpControllerState)
                | LEVEL(Value) | STEAM(Value)
                | REPAIRED(PhysicalUnit)
                | FAILURE_ACKNOWLEDGEMENT(PhysicalUnit)
                | junk;
end
```

For the received messages, in addition to the messages specified in Sec. C.12.6, we add an extra constant message *junk*. This message will represent any received message which does not belong to the class of recognized messages. We do not add a similar message to the messages sent, since we may assume that the steam-boiler control system will only send proper messages. Obviously, receiving a *junk* message will lead to the detection of a failure of the message transmission system.

In the SBCS_CST specification we describe the various constants that characterize the Steam-Boiler (see Sec. C.12.2).

spec SBCS_CST =
VALUE
then ops $C, M1, M2, N1, N2, W, U1, U2, P : Value$;
$dt : Value$ %% Time duration between two cycles (5 sec.)
%% These constants must verify some obvious properties:
• $0 < M1$ • $M1 < N1$ • $N1 < N2$ • $N2 < M2$ • $M2 < C$
• $0 < W$ • $0 < U1$ • $0 < U2$ • $0 < P$
end

We will also specify the datatypes "set of received messages" and "set of sent messages" since, as suggested at the end of Sec. C.12.3, all messages are supposed to be received (or emitted) simultaneously at each cycle. The two latter specifications are obtained by instantiation of a generic specification FINSET of "sets of elements". This generic specification FINSET is imported from the library BASIC/STRUCTUREDDATATYPES.

This leads to the following PRELIMINARY specification.

from BASIC/STRUCTUREDDATATYPES **get** FINSET

spec PRELIMINARY =
FINSET [MESSAGES_RECEIVED **fit** $Elem \mapsto R\_Message$]
and FINSET [MESSAGES_SENT **fit** $Elem \mapsto S\_Message$]
and SBCS_CST
end

C.3 Carrying On

As emphasized in Sec. C.12.3, the steam-boiler control system is a typical example of a control-command system. The specification of such systems always follows the same pattern:

- A preliminary set of specifications group all the basic information about the system to be controlled, such as the specification of the various messages to be exchanged between the system and its environment, and the specification of the various constants related to the system of interest. This is indeed the aim of the specification PRELIMINARY introduced in the previous section.
- Then, the various states of the control system should be described. At this stage, however, it would be much too early to determine which state variables are needed. Thus states will be represented by values of a (loosely specified) sort *State*, equipped with some observers (corresponding to access to state variables). During the requirements analysis and formalization phase we may need further observers, to be introduced on a by-need basis.
- Then a (group of) specification(s) will take care of the analysis of the received messages – here, of failure detection in particular. On the basis of

this analysis, some actions should be taken, corresponding to the messages to be sent to the environment. State variables are also updated according to the result of the analysis of the received messages and the messages to be sent.

- Finally a specification describes the overall control-command system as a labeled transition system.

A very rough preliminary sketch of the steam-boiler control system specification looks therefore as follows:

library SBCS
from BASIC/NUMBERS **get** NAT
from BASIC/STRUCTUREDDATATYPES **get** FINSET
%% Display annotations for half and square to be added here.

spec PRELIMINARY = %{ See previous section. }%

spec SBCS_STATE =
PRELIMINARY
then **sort** *State*
ops %% Needed state observers are introduced here.
%% E.g., we need an observer for the mode of operation:
mode : *State* → *Mode*;
...
end

spec SBCS_ANALYSIS =
SBCS_STATE
then %% Analysis of received messages and in particular failure detection.
%% Computation of the messages to be sent.
op *messages_to_send* : *State*×*FinSet*[*R_Message*] → *FinSet*[*S_Message*];
%% Computation of the updates of the state variables.
%{ For each observer *obs* defined in SBCS_STATE,
we introduce an operation *next_obs* that computes the
corresponding update according to the analysis of the messages
received in this round. For instance, we specify here an operation
next_mode corresponding to the update of the observer *mode*. }%
ops *next_mode* : *State* × *FinSet*[*R_Message*] → *Mode*;
...
end

spec STEAM_BOILER_CONTROL_SYSTEM =
SBCS_ANALYSIS
then **op** $init : State$
pred $is\_step : State \times FinSet[R\_Message] \times FinSet[S\_Message] \times State$
%% Specification of the initial state *init* by means of the observers, e.g:
- $mode(init) = Normal$
- ...

%{ Specification of *is_step* by means of the observers
and of the updating operations, e.g.: **}%**
$\forall s, s' : State;\ msgs : FinSet[R\_Message];\ Smsg : FinSet[S\_Message]$
- $is\_step(s, msgs, Smsg, s') \Leftrightarrow$
  $mode(s') = next\_mode(s, msgs) \wedge \ldots \wedge$
  $Smsg = messages\_to\_send(s, msgs)$

then **%%** Specification of the reachable states:
free { pred $reach : State$
$\forall s, s' : State;\ msgs : FinSet[R\_Message];\ Smsg : FinSet[S\_Message]$
- $reach(init)$
- $reach(s) \wedge is\_step(s, msgs, Smsg, s') \Rightarrow reach(s')$ **}**

end

Of course the specification SBCS_ANALYSIS is likely to be further structured into smaller pieces of specifications. Indeed, since the informal requirements are too complex to be handled as a whole, we will therefore successively concentrate on various parts of them. The study and formalization of each chunk of requirements will lead to specifications that will later on be put together to obtain the SBCS_ANALYSIS specification. As already pointed out, it is likely that when analyzing a chunk of requirements we will discover the need for new observers on states (i.e., new state variables). This means that the specification SBCS_STATE will be subject to iterated extensions where we introduce the new observers that are needed.

For instance, in Sec. C.12.6 it is explained that when the *STOP* message has been received three times in a row, the program must go into the *EmergencyStop* mode. We need therefore an observer (i.e., a state variable) to record the number of times we have successively received the *STOP* message. So in the sequel we will start from the following specification of states:

spec SBCS_STATE_1 =
PRELIMINARY
then **sort** *State*
ops $mode : State \to Mode$;
$nbSTOP : State \to Nat$

end

Introducing the new observer *nbSTOP* means that we will have to specify a corresponding *next_nbSTOP* operation in the SBCS_ANALYSIS specification.

C.4 Specifying the Mode of Operation

Our next step is the specification of the various operating modes in which the steam-boiler control system operates. (As explained in Sec. C.1 we do not take into account the *Initialization* mode in this specification.) According to Sec. C.12.4, the operating mode of the steam-boiler control system depends on which failures have been detected (see e.g. "all physical units [are] operating correctly", "a failure of the water level measuring unit", "detection of an erroneous transmission"). It depends also on the expected evolution of the water level (see "If the water level is risking...").

We will therefore assume that the specification SBCS_ANALYSIS will provide the following predicates which, given a known state and newly received messages, should reflect the failures detected by the steam-boiler control system:[4]

- $Transmission\_OK : State \times FinSet[R\_Message]$
  should hold iff we rely on the message transmission system,
- $PU\_OK : State \times FinSet[R\_Message] \times PhysicalUnit$
  should hold iff we rely on the corresponding physical unit,
- $DangerousWaterLevel : State \times FinSet[R\_Message]$
  should hold iff we estimate that the water level risks reaching the min (*M1*) or max (*M2*) limits.

However, at this stage our understanding of the steam-boiler control system is still quite preliminary, and it is therefore too early to attempt to specify these predicates. Therefore, our specification MODE_EVOLUTION, where we specify the new operating mode according to the previous one and the newly received messages (i.e., the operation *next_mode*), will be made *generic* w.r.t. these predicates. Let us emphasize that here genericity is used to ensure a *loose coupling* between the current specification of interest, MODE_EVOLUTION, and other specifications expected to provide the needed predicates.

Let us now explain how to specify the new mode of operation. At first glance the informal requirements (see Sec. C.12.4) look quite complicated, mainly because they explain, for each operating mode, under which conditions the steam-boiler control system should stay in the same operating mode or switch to another one. However, things get simpler if we analyze under which conditions the next mode is one of the specified operating modes. In particular, a careful analysis of the requirements shows that, except for switching to the *EmergencyStop* mode, we can determine the new operating mode (after receiving some messages) without even taking into account the previous one.

To improve the legibility of our specification it is better to introduce some auxiliary predicates (*Everything_OK*, *AskedToSop*, *SystemStillControllable*,

[4] It is important to make a subtle distinction between the actual failures, about which we basically know nothing, and the failures detected by the steam-boiler control system. In our specification, the behaviour of the steam-boiler control system is induced by the failures detected, whatever the actual failures are.

and *Emergency*) that will facilitate the characterization of the conditions under which the system switches from one mode to another:

- The aim of the predicate *Everything_OK* is to express that we believe that all physical units operate correctly, including the message transmission system.
- The aim of the predicate *AskedToSop* is to determine if we have received the *STOP* message three times in a row.
- The aim of the predicate *SystemStillControllable* is to characterize the conditions under which the steam-boiler control system will operate in *Rescue* mode. Let us point out that the corresponding part of the informal requirements (see Sec. C.12.4.4) is not totally clear, in particular the exact meaning of the sentence "if one can rely upon the information which comes from the units for controlling the pumps". There is a double ambiguity here: on the one hand it is unclear whether "the pumps" means "all pumps" or "at least one pump"; on the other hand there are two ways of "controlling" each pump (the information sent by the pump and the information sent by the pump controller), and it is unclear whether "controlling" refers to both of them or only to the pump controller. Our interpretation will be as follows: we consider it is enough that at least one pump is "correctly working", and for us correctly working will mean we rely on both the pump and the associated pump controller. As with all interpretations made during the formalization process, we should in principle interact with the designers of the informal requirements in order to clarify what was the exact intended meaning and to check that our interpretation is adequate. The important point is that our interpretation is entirely localized in the axiomatization of *SystemStillControllable*, and it will therefore be fairly easy to change our specification in case of misinterpretation.
- The aim of the predicate *Emergency* is to characterize when we should switch to the *EmergencyStop* mode. In Sec. C.12.4.2, it is said that the steam-boiler control system should switch from *Normal* mode to *Rescue* mode as soon as a failure of the water level measuring unit is detected. However, in Sec. C.12.4.4, it is explained that the steam-boiler control system can only operate in *Rescue* mode if some additional conditions hold (represented by our predicate *SystemStillControllable*). We decide therefore that when in *Normal* mode, if a failure of the water level measuring unit is detected, the steam-boiler control system will switch to *Rescue* mode only if *SystemStillControllable* holds, otherwise it will switch (directly) to *EmergencyStop* mode.[5]

[5] If our interpretation is incorrect, then in some cases we may have replaced a sequence *Normal* → *Rescue* → *EmergencyStop* by a sequence *Normal* → *EmergencyStop*. Note that a sequence *Normal* → *Rescue* → *Normal* or *Degraded* is not possible since several cycles are necessary between a failure detection and the decision that the corresponding unit is again fully operational, see Sec. C.5,

The axiomatization of the next mode of operation is now both simple and clear, as illustrated by the MODE_EVOLUTION specification.[6]

spec MODE_EVOLUTION
[**preds** $Transmission\_OK : State \times FinSet[R\_Message]$;
$PU\_OK : State \times FinSet[R\_Message] \times PhysicalUnit$;
$DangerousWaterLevel : State \times FinSet[R\_Message]$]
given SBCS_STATE_1 =
local %% Auxiliary predicates to structure the specification of *next_mode*.
preds $Everything\_OK, AskedToStop, SystemStillControllable,$
$Emergency : State \times FinSet[R\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message]$
- $Everything\_OK(s, msgs) \Leftrightarrow$
  $(Transmission\_OK(s, msgs) \wedge$
  $(\forall pu : PhysicalUnit \bullet PU\_OK(s, msgs, pu)))$
- $AskedToStop(s, msgs) \Leftrightarrow nbSTOP(s) = 2 \wedge STOP \in msgs$
- $SystemStillControllable(s, msgs) \Leftrightarrow$
  $(PU\_OK(s, msgs, OutputOfSteam) \wedge$
  $(\exists pn : PumpNumber \bullet PU\_OK(s, msgs, Pump(pn))$
  $\wedge PU\_OK(s, msgs, PumpController(pn))))$
- $Emergency(s, msgs) \Leftrightarrow$
  $(\ mode(s) = EmergencyStop \vee$
  $AskedToStop(s, msgs) \vee$
  $\neg\, Transmission\_OK(s, msgs) \vee$
  $DangerousWaterLevel(s, msgs) \vee$
  $(\neg\, PU\_OK(s, msgs, WaterLevel) \wedge$
  $\neg\, SystemStillControllable(s, msgs))\ )$

within ops $next\_mode : State \times FinSet[R\_Message] \to Mode;$
$next\_nbSTOP : State \times FinSet[R\_Message] \to Nat$
$\forall s : State;\ msgs : FinSet[R\_Message]$
%% Emergency stop mode:
- $Emergency(s, msgs) \Rightarrow next\_mode(s, msgs) = EmergencyStop$

%% Normal mode:
- $\neg\, Emergency(s, msgs) \wedge$
  $Everything\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Normal$

---

i.e., we must have a sequence of the form *Normal* → *Rescue* → ... → *Rescue* → *Normal* or *Degraded* in such cases.

[6] Note that once in the *EmergencyStop* mode, we specify that we stay in this mode forever, rather than specifying that the steam-boiler control system actually stops. Note also that we realize that the operation *next_nbSTOP* is better specified in this MODE_EVOLUTION specification.

%% Degraded mode:
- $\neg\, Emergency(s, msgs) \wedge$
  $\neg\, Everything\_OK(s, msgs) \wedge$
  $PU\_OK(s, msgs, WaterLevel) \wedge$
  $Transmission\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Degraded$

%% Rescue mode:
- $\neg\, Emergency(s, msgs) \wedge$
  $\neg\, PU\_OK(s, msgs, WaterLevel) \wedge$
  $SystemStillControllable(s, msgs) \wedge$
  $Transmission\_OK(s, msgs) \Rightarrow next\_mode(s, msgs) = Rescue$

%% *next_nbSTOP*:
- $next\_nbSTOP(s, msgs) = nbSTOP(s) + 1$ *when* $STOP \in msgs$
  else 0

end

In the next step of our formalization process, we will specify the predicates assumed by MODE_EVOLUTION, which amounts to specify the detection of equipment failures. This will be the topic of the next section.

C.5 Specifying the Detection of Equipment Failures

The detection of equipment failures is described in Sec. C.12.7. It is quite clear that this detection is the most difficult part to formalize, mainly because both our intuition and the requirements (see e.g. "knows from elsewhere", "incompatible with the dynamics") suggest that we should take into account some inter-dependencies when detecting the various possible failures.

For instance, if we ask a pump to stop, and if in the next cycle the pump state still indicates that the pump is open, we may in principle infer either a failure of the message transmission system (e.g. the stop order was not properly sent or was not received, or the message indicating the pump state has been corrupted) or a failure of the pump (which was not able to execute the stop order or which sends incorrect state messages). Our understanding of the requirements is that in such a case we must conclude there has been a failure of the pump, not of the message transmission system. Let us stress again that it is important to distinguish between the actual failures of the various pieces of equipment, and the diagnosis we will make. Only the latter is relevant in our specification.

C.5.1 Understanding the Detection of Equipment Failures

Before starting to specify the detection of equipment failures, we must proceed to a careful analysis of Sec. C.12.7, in order to clarify the inter-dependencies mentioned above. Only then will we be able to understand how to structure our specification of this crucial part of the problem.

A first rough analysis of the part of Sec. C.12.7, devoted to the description of potential failures of the physical units (i.e. of the pumps, the pump controllers and the two measuring devices) shows that these failures are detected on the basis of the information contained in the received messages: we must check that the received values are in accordance with some *expected* values (according to the history of the system, i.e. according to the "dynamics of the system" and to the messages previously sent by the steam-boiler control system). In particular, the detection of failures of the physical units relies on the fact that we have effectively received the necessary messages. If we have not received these messages, then we should conclude there has been a failure of the message transmission system (see below), and in these cases (see the MODE_EVOLUTION specification), the steam-boiler control system switches to the *EmergencyStop* mode. The further detection of failures of the physical units (in addition to the already detected failure of the message transmission system) is therefore irrelevant in such cases.

Let us now consider the message transmission system. The part of Sec. C.12.7 devoted to the description of potential failures of the message transmission system is quite short. Basically, it tells us that we should check that the steam-boiler control system has received all the messages it was expecting, and that none of the messages received is aberrant. However, it is important to note that the involved analysis of the received messages combines two aspects: on the one hand, there is some "static" analysis of the received messages in order to check that all messages that must be present in each transmission are effectively present (see Sec. C.12.6). These messages are exactly the messages required to proceed to the detection of the failures of the physical units. On the other hand, the steam-boiler control system expects to receive (or, on the contrary, not to receive) some specific messages according to the history of the system (for instance, the steam-boiler control system expects to receive a "failure acknowledgement" from a physical unit once it has detected a corresponding failure and sent a "failure" message to this unit, but not before), and here some "dynamic" analysis is required. Obviously, the "static" analysis of the messages can be made on the basis of the received messages only, while the "dynamic" analysis must take into account, in addition to the received messages, the history of the system, and more precisely the history of the failures detected so far and of the "failure acknowledgement" and "repaired" messages received so far.

From this first analysis we draw the following conclusions on how to specify the detection of equipment failures:

1. In a first step we should keep track of the failure status of the physical units. This will lead to a new observer *status* on states, and to a specification STATUS_EVOLUTION of how this status evolves, i.e., of a *next_status* operation.
2. Then we specify the detection of the message transmission system failures (hence *Transmission_OK*) in the specification MESSAGE_TRANSMISSION_

SYSTEM_FAILURE. As explained above, in a first step we take care of the "static" analysis of the received messages, and then in a second step we take care of the "dynamic" analysis of the received messages, using how we have kept track of the "status" of the physical units, i.e., using the observer *status*.

3. Then, for each physical unit, we specify the detection of its failures by comparing the received message with the *expected* one. For this comparison we can freely assume that the "static" analysis of the received messages has been successful, i.e., that the message sent by the physical unit has been received.

The corresponding specifications are described in the next subsections.

C.5.2 Keeping Track of the Status of the Physical Units

Remember that to perform the "dynamic" analysis of the received messages, as explained above, we must check that we receive "failure acknowledgement" and "repaired" messages when appropriate. In order to do this, we must keep track of the failures detected and of the "failure acknowledgement" and "repaired" messages received. Since the same reasoning applies for all physical units, we can do the analysis in a generic way. For each physical unit, we will keep track of its status, which can be either OK, $FailureWithoutAck$ or $FailureWithAck$. The status of a physical unit will then be updated accordingly to the detection of failures, and receipt of "failure acknowledgement" and "repaired" messages.

Thus, in a first step we should extend the specification SBCS_STATE_1 to add an observer related to the failure status of physical units:

spec SBCS_STATE_2 =
SBCS_STATE_1
then **free type** $Status ::= OK \mid FailureWithoutAck \mid FailureWithAck$
op $status : State \times PhysicalUnit \rightarrow Status$;
end

Now the specification of how the status of a physical unit evolves, i.e., of the operation *next_status* in STATUS_EVOLUTION, is quite straightforward. We rely again on the predicate PU_OK.[7]

[7] The reader may detect that the specification STATUS_EVOLUTION is not totally correct. However, we prefer to give here the text of the specification as it was originally written, and we will explain in Sec. C.9 how we detect, when validating the specification of the steam-boiler control system, that something is not correct, and how the problem can be fixed.

spec STATUS_EVOLUTION
[**pred** $PU\_OK : State \times FinSet[R\_Message] \times PhysicalUnit$]
given SBCS_STATE_2 =
op $next\_status : State \times FinSet[R\_Message] \times PhysicalUnit \rightarrow Status$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pu : PhysicalUnit$
- $status(s, pu) = OK \wedge PU\_OK(s, msgs, pu)$
  $\Rightarrow next\_status(s, msgs, pu) = OK$
- $status(s, pu) = OK \wedge \neg\, PU\_OK(s, msgs, pu)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$
- $status(s, pu) = FailureWithoutAck \wedge$
  $FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$
- $status(s, pu) = FailureWithoutAck \wedge$
  $\neg\,(FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithoutAck$
- $status(s, pu) = FailureWithAck \wedge REPAIRED(pu) \in msgs$
  $\Rightarrow next\_status(s, msgs, pu) = OK$
- $status(s, pu) = FailureWithAck \wedge \neg\,(REPAIRED(pu) \in msgs)$
  $\Rightarrow next\_status(s, msgs, pu) = FailureWithAck$

end

C.5.3 Detection of the Message Transmission System Failures

As explained above, we first specify the "static" analysis of the received messages, and then we specify the "dynamic" analysis of these messages.

To specify the "static" analysis of messages, it is necessary to check that all "indispensable" messages are present. In addition, a set of received messages is "acceptable" if there are no "duplicated" messages in this set. Since we have specified the collection of received messages as a set, we cannot have several occurrences of exactly the same message in this set. (Note that this means that our choice of using "sets" instead of "bags", for instance, is therefore not totally innocent: either we assume that receiving several occurrences of exactly the same message will never happen, and this is an assumption on the environment, or we assume that this case should not lead to the detection of a failure of the message transmission system, and this is an assumption on the requirements.) However, specifying the collection of received messages as a set does not imply that a set of received messages cannot contain several $LEVEL(v)$ messages, with distinct values (for instance). Hence we must check this explicitly.

Remember that receiving "unknown" messages (i.e., messages that do not belong to the list of messages as specified in Sec. C.12.6) is taken into account via the extra constant *junk* message (see the specification MESSAGES_RECEIVED). We believe also that we cannot simultaneously receive a failure acknowledgement and a repaired message for the same physical unit, i.e., that

at least one cycle is needed between acknowledging the failure and repairing the unit. We will check this as well.[8]

We focus now on the "dynamic" analysis of the received messages. As explained above, to perform this "dynamic" analysis, we check that we receive "failure acknowledgement" and "repaired" messages when appropriate, according to the current status of each physical unit. We understand that for each failure signaled by the steam-boiler control system, the corresponding physical unit will send just one failure acknowledgement. Moreover, we will specify the steam-boiler control system in such a way that when it receives a "repaired" message, the steam-boiler control system acknowledges it immediately. Hence, if there is no problem with the message transmission system, and due to the fact that transmission time can be neglected, the steam-boiler control system must in principle receive only one repaired message for a given failure. Note that this is not contradictory with the "until..." part of the sentences describing the "repaired" messages in the informal requirements (cf. Sec. C.12.6). To summarize, we consider we receive an unexpected message when:

- the program receives initialization messages but is no longer in initialization mode, or
- the program receives for some physical unit a "failure acknowledgement" without having previously sent the corresponding failure detection message, or receives redundant failure acknowledgements, or
- the program receives for some physical unit a "repaired message", but the unit is OK or its failure is not yet acknowledged.

We now have all the ingredients required to specify the *Transmission_OK* predicate, taking into account both static and dynamic aspects, which leads to the following MESSAGE_TRANSMISSION_SYSTEM_FAILURE specification.

[8] We must confess that this belief is induced by our intuition about the behaviour of the system. Indeed nothing in the requirements allows us to make either this interpretation or the opposite one. Although not essential, this assumption will simplify the axiomatization.

spec MESSAGE_TRANSMISSION_SYSTEM_FAILURE =
SBCS_STATE_2
then **local** %% Static analysis:
pred $\_\_is\_static\_OK : FinSet[R\_Message]$
$\forall msgs : FinSet[R\_Message]$
• $msgs\ is\_static\_OK \Leftrightarrow$
$(\ \neg(junk \in msgs) \wedge$
$(\exists! v : Value \bullet LEVEL(v) \in msgs) \wedge$
$(\exists! v : Value \bullet STEAM(v) \in msgs) \wedge$
$(\forall pn : PumpNumber \bullet \exists! ps : PumpState \bullet$
$PUMP\_STATE(pn, ps) \in msgs) \wedge$
$(\forall pn : PumpNumber \bullet \exists! pcs : PumpControllerState \bullet$
$PUMP\_CONTROLLER\_STATE(pn, pcs) \in msgs) \wedge$
$(\forall pu : PhysicalUnit \bullet$
$\neg\,(FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs$
$\wedge\ REPAIRED(pu) \in msgs))\ )$
%% Dynamic analysis:
pred $\_\_is\_NOT\_dynamic\_OK\_for\_\_ : FinSet[R\_Message] \times State$
$\forall s : State;\ msgs : FinSet[R\_Message]$
• $msgs\ is\_NOT\_dynamic\_OK\_for\ s \Leftrightarrow$
$(\ (\ \neg(mode(s) = Initialization) \wedge$
$(\ STEAM\_BOILER\_WAITING \in msgs \vee$
$PHYSICAL\_UNITS\_READY \in msgs\ )\ )$
$\vee\ (\ \exists pu : PhysicalUnit \bullet$
$FAILURE\_ACKNOWLEDGEMENT(pu) \in msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithAck)\ )$
$\vee\ (\ \exists pu : PhysicalUnit \bullet$
$REPAIRED(pu) \in msgs \wedge$
$(status(s, pu) = OK \vee status(s, pu) = FailureWithoutAck)\ ))$
within
pred $Transmission\_OK : State \times FinSet[R\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message]$
• $Transmission\_OK(s, msgs) \Leftrightarrow$
$(msgs\ is\_static\_OK \wedge \neg(msgs\ is\_NOT\_dynamic\_OK\_for\ s))$
end

C.5.4 Detection of the Pump and Pump Controller Failures

We start by considering the detection of the failures of the pumps.

As explained in Sec C.5.1, we rely on the predicted pump state message. Thus, in a first step we should extend the specification SBCS_STATE_2 to add an observer related to the prediction of pump state messages. The prediction (*Open* or *Closed*) can however only be made when the status of the corresponding pump is *OK*. This is why we extend the sort *PumpState* to introduce a constant *Unknown_PS*:

spec SBCS_STATE_3 =
SBCS_STATE_2
then **free type** *ExtendedPumpState* ::= *sort PumpState* | *Unknown_PS*
op *PS_predicted* : *State* × *PumpNumber* → *ExtendedPumpState*;
%{ $status(s, Pump(pn)) = OK \Leftrightarrow$
$\neg\,(PS\_predicted(s, pn) = Unknown\_PS)$ **}%**
end

The specification of the detection of pump failures is now straightforward and is given in the PUMP_FAILURE specification. Remember that the meaning of *Pump_OK* is only relevant when *Transmission_OK* holds, which in particular implies that for each pump, there is only one *PUMP_STATE* message for it in *msgs*. Moreover, we check the received value only if the predicted value is not *Unknown_PS*.

spec PUMP_FAILURE =
SBCS_STATE_3
then **pred** *Pump_OK* : *State* × *FinSet*[*R_Message*] × *PumpNumber*
$\forall s$: *State*; *msgs* : *FinSet*[*R_Message*]; *pn* : *PumpNumber*
• $Pump\_OK(s, msgs, pn) \Leftrightarrow$
$PS\_predicted(s, pn) = Unknown\_PS \vee$
$PUMP\_STATE(pn, PS\_predicted(s, pn)\ as\ PumpState) \in msgs$
end

Let us now consider the detection of the failures of the pump controllers. Again we rely on the predicted pump state controller message. Here, we must be a bit careful in order to reflect the fact that stopping a pump has an instantaneous effect, while starting it takes five seconds (see Sec. C.12.2.3). Since five seconds is, unfortunately, exactly the elapsed time between two cycles, when we decide to activate a pump we may have to wait two cycles to receive a corresponding *Flow* pump controller state. This is why, in addition to the constant *Unknown_PCS*, used for the cases where no prediction can be made since the pump controller is not working correctly, we also introduce a constant *SoonFlow* to be used for the prediction related to a just activated pump:

spec SBCS_STATE_4 =
SBCS_STATE_3
then **free type**
ExtendedPumpControllerState ::= *sort PumpControllerState*
| *SoonFlow* | *Unknown_PCS*
op *PCS_predicted* : *State* × *PumpNumber*
→ *ExtendedPumpControllerState*;
%{ $status(s, PumpController(pn)) = OK \Rightarrow$
$\neg\,(PCS\_predicted(s, pn) = Unknown\_PCS)$ **}%**
end

The specification of the detection of pump controller failures is now straightforward and is given in the PUMP_CONTROLLER_FAILURE specification. Remember that the meaning of *Pump_Controller_OK* is only relevant

when *Transmission_OK* holds, which in particular implies that for each pump, there is only one *PUMP_CONTROLLER_STATE* message for it in *msgs*. Moreover, we check the received value only if the predicted value is either *Flow* or *NoFlow*, since if it is *SoonFlow* or *Unknown_PCS* we cannot conclude.

spec PUMP_CONTROLLER_FAILURE =
SBCS_STATE_4
then **pred** $Pump\_Controller\_OK : State \times FinSet[R\_Message] \times PumpNumber$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber$
• $Pump\_Controller\_OK(s, msgs, pn) \Leftrightarrow$
$PCS\_predicted(s, pn) = Unknown\_PCS$
$\vee\ PCS\_predicted(s, pn) = SoonFlow$
$\vee\ PUMP\_CONTROLLER\_STATE(pn,$
$PCS\_predicted(s, pn)\ as\ PumpControllerState) \in msgs$
end

C.5.5 Detection of the Steam and Water Level Measurement Device Failures

To specify the failures of the steam and water level measurement devices, we must again rely on some predicted values. Here however we cannot predict an exact value, but only an interval in which the received value should be contained. This leads to the following extension of SBCS_STATE_4:

spec SBCS_STATE_5 =
SBCS_STATE_4
then **free type** $Valpair ::= pair(low : Value;\ high : Value)$
ops $steam\_predicted, level\_predicted : State \to Valpair$;
%{ $low(steam\_predicted(s))$ is the minimal output of steam predicted, $high(steam\_predicted(s))$ is the maximal output of steam predicted, and similarly for *level_predicted*. **}%**
end

The specification of the failures of the measurement devices is again straightforward and is given in the STEAM_FAILURE and LEVEL_FAILURE specifications. Remember that the meaning of *Steam_OK* (*Level_OK* resp.) is only relevant when *Transmission_OK* holds, which in particular implies that there is only one $STEAM(v)$ ($LEVEL(v)$ resp.) message in *msgs* (hence only one possible v in the quantifications $\forall v : Value\ \ldots$ below). Note also that here we assume that the predicted values will take care of the static limits (*0* and W for the steam, *0* and C for the water level), thus we do not need to check these static limits explicitly here.

spec STEAM_FAILURE =
SBCS_STATE_5
then **pred** $Steam\_OK : State \times FinSet[R\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message]$
• $Steam\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value \bullet STEAM(v) \in msgs \Rightarrow$
$(low(steam\_predicted(s)) \leq v) \wedge$
$(v \leq high(steam\_predicted(s)))\ )$
end

spec LEVEL_FAILURE =
SBCS_STATE_5
then **pred** $Level\_OK : State \times FinSet[R\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message]$
• $Level\_OK(s, msgs) \Leftrightarrow$
$(\forall v : Value \bullet LEVEL(v) \in msgs \Rightarrow$
$(low(level\_predicted(s)) \leq v) \wedge$
$(v \leq high(level\_predicted(s)))\ )$
end

C.5.6 Summing Up

We now have all the ingredients necessary for the specification of the predicate PU_OK. This is done in the FAILURE_DETECTION specification, which integrates together all the specifications related to failure detection.

spec FAILURE_DETECTION =
MESSAGE_TRANSMISSION_SYSTEM_FAILURE
and PUMP_FAILURE **and** PUMP_CONTROLLER_FAILURE
and STEAM_FAILURE **and** LEVEL_FAILURE
then **pred** $PU\_OK : State \times FinSet[R\_Message] \times PhysicalUnit$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber$
• $PU\_OK(s, msgs, Pump(pn)) \Leftrightarrow Pump\_OK(s, msgs, pn)$
• $PU\_OK(s, msgs, PumpController(pn)) \Leftrightarrow$
$Pump\_Controller\_OK(s, msgs, pn)$
• $PU\_OK(s, msgs, OutputOfSteam) \Leftrightarrow Steam\_OK(s, msgs)$
• $PU\_OK(s, msgs, WaterLevel) \Leftrightarrow Level\_OK(s, msgs)$
hide **ops** $Pump\_OK,\ Pump\_Controller\_OK,\ Steam\_OK,\ Level\_OK$
end

C.6 Predicting the Behaviour of the Steam-Boiler

In the previous section we have explained that failure detection was to a large extent based on a comparison between the received messages and the expected

ones. For this purpose we have extended the specification SBCS_STATE by several observers, which means we have assumed that each cycle, we record in some state variables the information needed to compute the expected messages at the next cycle. According to our explanations in Sec. C.3, we must now specify, for each observer *obs* introduced, a corresponding *next_obs* operation. This is the aim of this section.

We have already defined the operation *next_mode* in the generic specification MODE_EVOLUTION (see Sec. C.4) and the operation *next_status* in the generic specification STATUS_EVOLUTION (see Sec. C.5.2). Thus what is left is the specification of the operations *next_PS_predicted*, *next_PCS_predicted*, *next_steam_predicted* and *next_level_predicted*.

As explained in Sec. C.5, the informal requirements suggest that we should take into account some inter-dependencies when predicting values to be received at the next cycle. For instance, the water level in the steam-boiler depends on how much steam is produced, but also on how much water is poured into the steam boiler by the pumps which are open. The information provided by the water level prediction is obviously crucial to decide whether we should activate or stop some pumps. On the other hand, to predict the pump state and pump controller state messages to be received at the next cycle, we must know which pumps have been ordered to be activated or to be stopped.

From this first analysis we draw the following conclusions on how to specify the needed predictions:

1. In a first step we should predict the interval in which the output of steam is expected to stay during the next cycle: this prediction relies only on the just received value $STEAM(v)$ (if we trust it) or on the previously predicted values for the steam production. This is because the production of steam is expected to vary according to its maximum gradients of increase and decrease, and nothing else.
2. In the next step we should decide whether some pumps have to be ordered to activate or to stop. This decision, plus the knowledge about the current state of the pumps (as much as we trust it), and the predicted evolution of the steam production, should allow us to predict the evolution of the water level.
3. Then, on the basis of the current states of the pumps and pump controllers, together with the above made choice of pumps to be activated or stopped, we can predict the states of the pumps and of the pump controllers at the next cycle.

Of course all these predictions are only meaningful as long as no failure of the message transmission system has been detected (but if this is not the case the steam-boiler control system switches to the *EmergencyStop* mode and stops, so no predictions are needed anyway). The corresponding specifications are described in the next subsections.

C.6.1 Predicting the Output of Steam and the Water Level

To predict the intervals in which the output of steam and the water level are expected to stay during the next cycle, we will proceed as follows:

1. Following the analysis sketched above, when we are in the state s and have received the messages *msgs*, to predict the interval in which the output of steam is expected to stay during the next cycle, we first should compute the *adjusted_steam* interval: this interval is either the (interval reduced to the) *received_steam* value if we can rely on it (i.e., if $PU\_OK(s, msgs, OutputOfSteam)$ holds), or the *steam_predicted* interval (stored in the state s at the previous cycle).
2. Then, we use the maximum gradients of increase and decrease (i.e., *U1* and *U2*), to predict the interval in which the output of steam is expected to stay during the next cycle. For this we use two auxiliary operations, *low_predicted_steam* and *high_predicted_steam*, to compute the lower and upper bounds of this interval.
3. We proceed similarly for the water level: first we compute the *adjusted_level* interval, which is either the (interval reduced to the) *received_level* value if we can rely on it (i.e., if $PU\_OK(s, msgs, WaterLevel)$ holds), or the *level_predicted* interval (stored in the state s at the previous cycle).
4. Then we should consider *broken_pumps* (the pumps pn for which neither $PU\_OK(s, msgs, Pump(pn))$ nor $PU\_OK(s, msgs, PumpController(pn))$ holds) and the *reliable_pumps*, which are not broken and are therefore known to be either *Open* or *Closed*.
5. At this point we must decide which pumps are ordered to activate or to stop.
   However, the specific control strategy for deciding which pumps should be activated or stopped need not to be detailed in this requirements specification: this can be left to a further refinement towards an implementation of the steam-boiler control system. (Obviously the strategy should compare the *adjusted_level* with the recommended interval $(N1, N2)$ and decide accordingly.)
   We will therefore rely on a loosely specified *chosen_pumps* operation, for which we just impose some soundness conditions (e.g., a pump ordered to activate should be currently considered as "reliable" and *Closed*, a pump ordered to stop should be currently considered as "reliable" and *Open*).
6. Now we can compute the minimal and maximal amounts of water that will be poured into the steam-boiler during the next cycle. To compute *minimal_pumped_water*, we consider that only the pumps which are "reliable" and already *Open* will pour some water in; the *broken_pumps*, the pumps which are just ordered to activate, and the pumps which are ordered to stop are all considered not pouring water in. Similarly, to compute *maximal_pumped_water*, we consider that the pumps which are "reliable" and already *Open*, the pumps which are just ordered to activate, as well as all the *broken_pumps*, may pour some water in; only the "reliable" pumps

just ordered to stop or already stopped are known not to be pouring any water in.

7. Finally, we can predict the interval in which the water level is expected to stay during the next cycle. For this we use two auxiliary operations, *low_predicted_level* and *high_predicted_level*, to compute the lower and upper bounds of this interval.
8. This prediction is the basis for deciding whether the water level risks to reach a *DangerousWaterLevel* (i.e., below *M1* or above *M2*).

Note that the intervals in which the output of steam and the water level are expected to stay during the next cycle are predicted without considering the *next_status* of these devices. This is indeed necessary for the *Degraded* and *Rescue* operating modes. This leads to the following STEAM_AND_LEVEL_PREDICTION specification.

spec STEAM_AND_LEVEL_PREDICTION =
FAILURE_DETECTION **and** FINSET [**sort** *PumpNumber*]
then **local**
ops $received\_steam : State \times FinSet[R\_Message] \to Value$;
$adjusted\_steam : State \times FinSet[R\_Message] \to Valpair$;
$low\_predicted\_steam, high\_predicted\_steam :$
$State \times FinSet[R\_Message] \to Value$;
$received\_level : State \times FinSet[R\_Message] \to Value$;
$adjusted\_level : State \times FinSet[R\_Message] \to Valpair$;
$low\_predicted\_level, high\_predicted\_level :$
$State \times FinSet[R\_Message] \to Value$;
$broken\_pumps : State \times FinSet[R\_Message] \to FinSet[PumpNumber]$;
$reliable\_pumps :$
$State \times FinSet[R\_Message] \times PumpState \to FinSet[PumpNumber]$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber;\ ps : PumpState$
%% Axioms for STEAM:
- $Transmission\_OK(s, msgs) \Rightarrow$
$STEAM(received\_steam(s, msgs)) \in msgs$
- $adjusted\_steam(s, msgs) =$
$pair(received\_steam(s, msgs), received\_steam(s, msgs))$
$when\ (Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, OutputOfSteam))$
$else\ steam\_predicted(s)$

%% Axioms for LEVEL:
- $Transmission\_OK(s, msgs) \Rightarrow$
$LEVEL(received\_level(s, msgs)) \in msgs$
- $adjusted\_level(s, msgs) =$
$pair(received\_level(s, msgs), received\_level(s, msgs))$
$when\ (Transmission\_OK(s, msgs) \wedge PU\_OK(s, msgs, WaterLevel))$
$else\ level\_predicted(s)$

%% Axioms for auxiliary pumps operations:

- $pn \in broken\_pumps(s, msgs) \Leftrightarrow$
  $\neg (\ PU\_OK(s, msgs, Pump(pn)) \wedge$
  $PU\_OK(s, msgs, PumpController(pn))\ )$
- $pn \in reliable\_pumps(s, msgs, ps) \Leftrightarrow$
  $\neg (pn \in broken\_pumps(s, msgs)) \wedge$
  $PUMP\_STATE(pn, ps) \in msgs$

within

ops $next\_steam\_predicted : State \times FinSet[R\_Message] \rightarrow Valpair$;
$chosen\_pumps :$
$State \times FinSet[R\_Message] \times PumpState \rightarrow FinSet[PumpNumber]$;
%{ $minimal\_pumped\_water$ and $maximal\_pumped\_water$ cannot be made local since their specification relies on $chosen\_pumps$ which must be exported. **}%**
$minimal\_pumped\_water, maximal\_pumped\_water :$
$State \times FinSet[R\_Message] \rightarrow Value$;
$next\_level\_predicted : State \times FinSet[R\_Message] \rightarrow Valpair$

pred $DangerousWaterLevel : State \times FinSet[R\_Message]$

%% Axioms for STEAM:

$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber$

- $low(next\_steam\_predicted(s, msgs)) =$
  $max(0, low(adjusted\_steam(s, msgs)) - (U2 \times dt))$
- $high(next\_steam\_predicted(s, msgs)) =$
  $min(W, high(adjusted\_steam(s, msgs)) + (U1 \times dt))$

%% Axioms for PUMPS:

- $pn \in chosen\_pumps(s, msgs, Open) \Rightarrow$
  $pn \in reliable\_pumps(s, msgs, Closed)$
- $pn \in chosen\_pumps(s, msgs, Closed) \Rightarrow$
  $pn \in reliable\_pumps(s, msgs, Open)$
- $minimal\_pumped\_water(s, msgs) =$
  $dt \times P \times \sharp(reliable\_pumps(s, msgs, Open)$
  $- chosen\_pumps(s, msgs, Closed))$
- $maximal\_pumped\_water(s, msgs) =$
  $dt \times P \times \sharp((reliable\_pumps(s, msgs, Open)$
  $\cup chosen\_pumps(s, msgs, Open)$
  $\cup broken\_pumps(s, msgs))$
  $- chosen\_pumps(s, msgs, Closed))$

%% Axioms for LEVEL:

- $low(next\_level\_predicted(s, msgs)) =$
  $max(0,\ (low(adjusted\_level(s, msgs))$
  $+ minimal\_pumped\_water(s, msgs))$
  $- (\ (dt^2 \times U1/2)$
  $+ (dt \times high(adjusted\_steam(s, msgs)))\ )\ )$

- $high(next\_level\_predicted(s, msgs)) =$
  $min(C,\ (high(adjusted\_level(s, msgs))$
  $+ maximal\_pumped\_water(s, msgs))$
  $-\ (\ (dt^2 \times U2/2)$
  $+\ (dt \times low(adjusted\_steam(s, msgs)))\ )\ )$
- $DangerousWaterLevel(s, msgs) \Leftrightarrow$
  $(low(next\_level\_predicted(s, msgs)) \leq M1) \vee$
  $(M2 \leq high(next\_level\_predicted(s, msgs)))$

hide **ops** *minimal_pumped_water*, *maximal_pumped_water*
end

C.6.2 Predicting the Pump and Pump Controller States

Specifying the predicted state of each pump at the next cycle is almost trivial. The next pump state is *Unknown_PS* if the *next_status* of the pump is not *OK*, otherwise it should be *Open* if:

- it is *Open* now and the pump is not ordered to stop, or
- the pump is ordered to activate;

otherwise, it should be *Closed* since:

- it is *Closed* now and the pump is not ordered to activate, or
- it is ordered to stop.

This leads to the following PUMP_STATE_PREDICTION specification. This specification extends STEAM_AND_LEVEL_PREDICTION (since we rely on *chosen_pumps* for our predictions), and STATUS_EVOLUTION (which provides *next_status*) *instantiated* by FAILURE_DETECTION (which provides the predicate *PU_OK* parameter of FAILURE_DETECTION).

spec PUMP_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then **op** $next\_PS\_predicted :$
$State \times FinSet[R\_Message] \times PumpNumber \rightarrow ExtendedPumpState$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber$
- $next\_PS\_predicted(s, msgs, pn) =$
  $Unknown\_PS\ when\ \neg(next\_status(s, msgs, Pump(pn)) = OK)$
  $else\ Open\ when\ (\ PUMP\_STATE(pn, Open)\ \epsilon\ msgs \wedge$
  $\neg(pn\ \epsilon\ chosen\_pumps(s, msgs, Closed))\ )$
  $else\ Closed$

end

The reasoning to predict the pump controller state is similar, but we must take into account that two cycles may be needed before a just activated pump leads to a *Flow* state (provided the pump is not stopped meanwhile). Thus, the next pump controller state is *Unknown_PCS* if the *next_status* of the

pump controller is not *OK*, or if the *next_status* of the corresponding pump is not *OK*, otherwise the predicted pump controller state value is:

- *Flow* when the pump is not ordered to stop and it is currently *Flow*, or it is currently *NoFlow* but *PCS_predicted SoonFlow*;
- *NoFlow* if the pump is ordered to stop, or if it is currently *NoFlow* and is not *PCS_predicted SoonFlow* and the pump is not ordered to activate;
- *SoonFlow* otherwise.

This leads to the following PUMP_CONTROLLER_STATE_PREDICTION specification.

spec PUMP_CONTROLLER_STATE_PREDICTION =
STATUS_EVOLUTION [FAILURE_DETECTION]
and STEAM_AND_LEVEL_PREDICTION
then **op** *next_PCS_predicted* :
$State \times FinSet[R\_Message] \times PumpNumber \rightarrow ExtendedPumpControllerState$
$\forall s : State;\ msgs : FinSet[R\_Message];\ pn : PumpNumber$
• $next\_PCS\_predicted(s, msgs, pn) =$
Unknown_PCS when
$\neg(\ next\_status(s, msgs, PumpController(pn)) = OK \wedge$
$next\_status(s, msgs, Pump(pn)) = OK\ )$
else Flow when
$(\ PUMP\_CONTROLLER\_STATE(pn, Flow) \in msgs \vee$
$(\ PUMP\_CONTROLLER\_STATE(pn, NoFlow) \in msgs \wedge$
$PCS\_predicted(s, pn) = SoonFlow\ )\ )$
$\wedge \neg(pn \in chosen\_pumps(s, msgs, Closed))$
else NoFlow when
$(pn \in chosen\_pumps(s, msgs, Closed))$
$\vee\ (\ PUMP\_CONTROLLER\_STATE(pn, NoFlow) \in msgs \wedge$
$\neg(PCS\_predicted(s, pn) = SoonFlow) \wedge$
$\neg(pn \in chosen\_pumps(s, msgs, Open))\ )$
else SoonFlow
end

All our predictions are summarized in the following PU_PREDICTION specification:

spec PU_PREDICTION =
PUMP_STATE_PREDICTION
and PUMP_CONTROLLER_STATE_PREDICTION
%{ Both specifications extend STATUS_EVOLUTION
(instantiated by FAILURE_DETECTION)
and STEAM_AND_LEVEL_PREDICTION **}%**
end

C.7 Specifying the Messages to Send

At this stage we are left with the specification of the messages to send at each cycle. This is easily specified, following Sec. C.12.5, and leads to the following SBCS_ANALYSIS specification.

The specification SBCS_ANALYSIS is obtained by instantiating the MODE_EVOLUTION specification by PU_PREDICTION, and extending the result by the specification of the operation *messages_to_send*.

spec SBCS_ANALYSIS =
MODE_EVOLUTION [PU_PREDICTION]
then **local**
ops $PumpMessages, FailureDetectionMessages :$
$State \times FinSet[R\_Message] \to FinSet[S\_Message];$
$RepairedAcknowledgementMessages :$
$FinSet[R\_Message] \to FinSet[S\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message];\ Smsg : S\_Message$
• $Smsg \in PumpMessages(s, msgs) \Leftrightarrow$
$(\exists pn : PumpNumber \bullet$
$(\ pn \in chosen\_pumps(s, msgs, Open)$
$\wedge\ Smsg = OPEN\_PUMP(pn)\ )$
$\vee\ (\ pn \in chosen\_pumps(s, msgs, Closed)$
$\wedge\ Smsg = CLOSE\_PUMP(pn)\ )\ )$
• $Smsg \in FailureDetectionMessages(s, msgs) \Leftrightarrow$
$(\exists pu : PhysicalUnit \bullet$
$Smsg = FAILURE\_DETECTION(pu) \wedge$
$next\_status(s, msgs, pu) = FailureWithoutAck\ )$
• $Smsg \in RepairedAcknowledgementMessages(msgs) \Leftrightarrow$
$(\exists pu : PhysicalUnit \bullet$
$Smsg = REPAIRED\_ACKNOWLEDGEMENT(pu) \wedge$
$next\_status(s, msgs, pu) = FailureWithAck\ )$
within
op $messages\_to\_send : State \times FinSet[R\_Message] \to FinSet[S\_Message]$
$\forall s : State;\ msgs : FinSet[R\_Message]$
• $messages\_to\_send(s, msgs) =$
$(PumpMessages(s, msgs)\ \cup$
$FailureDetectionMessages(s, msgs)\ \cup$
$RepairedAcknowledgementMessages(msgs))$
$+\ MODE(next\_mode(s, msgs))$
end

C.8 The Steam-Boiler Control System Specification

According to our work plan detailed in Sec. C.3, we have already specified the main parts of our case study. First, let us display a basic (flat) specification equivalent to SBCS_STATE_5 and where all the state observers are listed together.

spec SBCS_STATE =
PRELIMINARY
then **sort** *State*
free type $Status ::= OK \mid FailureWithoutAck \mid FailureWithAck$
free type $ExtendedPumpState ::= sort\ PumpState \mid Unknown\_PS$
free type
$ExtendedPumpControllerState ::= sort\ PumpControllerState \mid SoonFlow \mid Unknown\_PCS$
free type $Valpair ::= pair(low : Value;\ high : Value)$
ops $mode : State \to Mode$;
$nbSTOP : State \to Nat$;
$status : State \times PhysicalUnit \to Status$;
$PS\_predicted : State \times PumpNumber \to ExtendedPumpState$;
$PCS\_predicted : State \times PumpNumber \to ExtendedPumpControllerState$;
$steam\_predicted, level\_predicted : State \to Valpair$
end

Perhaps a better alternative is to choose arbitrary values and not bother the reader with this initialization phase.

We are now ready to provide the specification of the steam-boiler control system, considered as a labeled transition system. We leave partly unspecified the initial state *init*, since in our specification this state represents the state immediately following the receipt of the *PHYSICAL_UNITS_READY* message. Hence intuitively the omitted axioms should take into account the messages sent and received during the initialization phase (at least at the end of it). It is therefore better to leave open for now the value of $nbSTOP(init)$, $status(init)$, $PS\_predicted(init, pn)$, $PCS\_predicted(init, pn)$, $steam\_predicted(init)$, and $level\_predicted(init)$, and to note that this would have to be taken care of when specifying the initialization phase. The value of $mode(init)$ is specified according to the end of Sec. C.12.4.1.

spec STEAM_BOILER_CONTROL_SYSTEM =
SBCS_ANALYSIS
then **op** $init : State$
pred $is\_step : State \times FinSet[R\_Message] \times FinSet[S\_Message] \times State$
%% Specification of the initial state *init*:
- $mode(init) = Normal \vee mode(init) = Degraded$

%% Specification of *is_step*:
$\forall s, s' : State;\ msgs : FinSet[R\_Message];\ Smsg : FinSet[S\_Message]$
• $is\_step(s, msgs, Smsg, s') \Leftrightarrow$
$mode(s') = next\_mode(s, msgs) \wedge$
$nbSTOP(s') = next\_nbSTOP(s, msgs) \wedge$
$(\forall pu : PhysicalUnit \bullet status(s', pu) = next\_status(s, msgs, pu)) \wedge$
$(\forall pn : PumpNumber \bullet$
$PS\_predicted(s', pn) = next\_PS\_predicted(s, msgs, pn) \wedge$
$PCS\_predicted(s', pn) = next\_PCS\_predicted(s, msgs, pn)\ ) \wedge$
$steam\_predicted(s') = next\_steam\_predicted(s, msgs) \wedge$
$level\_predicted(s') = next\_level\_predicted(s, msgs) \wedge$
$Smsg = messages\_to\_send(s, msgs)$

then %% Specification of the reachable states:
free { **pred** $reach : State$
$\forall s, s' : State;\ msgs : FinSet[R\_Message];\ Smsg : FinSet[S\_Message]$
• $reach(init)$
• $reach(s) \wedge is\_step(s, msgs, Smsg, s') \Rightarrow reach(s')$ **}**

end

C.9 Validation of the CASL Requirements Specification

Once the formalization of the informal requirements is completed, we must now face the following question: is our formal specification adequate ? Answering this question is a difficult issue since there is no formal way to establish the adequacy of a formal specification w.r.t. informal requirements, i.e., we cannot *prove* this adequacy. However, we can try to *test* it, by performing various "experiments". When these experiments are successful, we increase our confidence in the formal specification. If some experiment fails, then we can inspect the specification and try to understand the causes of the failure, possibly detecting some flaw in the specification.

We will base our validation process on theorem proving, i.e., we will check that some formulas are logical consequences of our requirements specification Steam_Boiler_Control_System. For this purpose we use the tools described in Chap. 10. During this validation process we can consider two kinds of proof obligations:

1. We can inspect the text of the specification and derive from this inspection some formulae that are expected to be logical consequences of our specification. This can be considered as some kind of "internal validation" of the formal specification.
2. We can check that some expected properties inferred from the informal requirements are logical consequences of our specification ("external validation"). To do this, we must first reanalyze the informal specification, state some expected properties, translate them into formulas, and then

attempt to prove that these formulas are logical consequences of our specification. This task is not easy, since in general one has the feeling that all expected properties were already detected and included in the axioms during the formalization process.

The application of these principles to the requirements specification of the steam-boiler control system leads to various proofs. Below we give only a few illustrative examples.

For instance, let us consider the specification of *next_mode* in MODE_EVOLUTION: it is advisable to prove that all the cases considered in the specification of *next_mode* are mutually exclusive, and that their disjunction is equivalent to true. This is a typical example of "internal validation" of the specification, since we just consider the text of the specification to decide which proof attempt will be performed, without considering the informal requirements again. We do not spell out the corresponding proofs here, but the reader can easily check that indeed the operation *next_mode* is well-defined (i.e., all cases are mutually exclusive and their disjunction is equivalent to true). In the same spirit we can prove that the same pump cannot be simultaneously be ordered to activate and to stop, that we never resignal a failure which has already been signaled, that as long as the operating mode is not set to *EmergencyStop* the water level is safe, etc.

Let us now consider an example of "external validation". According to our understanding of failure detection (see Sec. C.5), if we have detected a failure of some physical unit *pu* (so *PU_OK* does not hold for *pu*), then the status of this physical unit should not be set to *OK*. The corresponding proof obligation reads as follows:

$$\begin{array}{l}\text{STEAM\_BOILER\_CONTROL\_SYSTEM} \models \\ \quad \forall s : State;\ msgs : FinSet[R\_Message];\ pu : PhysicalUnit \\ \quad \bullet\ Transmission\_OK(s, msgs) \wedge \neg\, PU\_OK(s, msgs, pu) \\ \qquad \Rightarrow \neg\,(next\_status(s, msgs, pu) = OK)\end{array}$$

However here we are unable to discharge this proof obligation. A careful analysis of the proof attempt shows that the proof fails since it could be the case that, simultaneously with the receipt of a repaired message for the physical unit *pu*, we nevertheless detect again a failure of the same unit. From this analysis we conclude that the following axiom in STATUS_EVOLUTION is not adequate:

$$\begin{array}{l}\bullet\ status(s, pu) = FailureWithAck \wedge REPAIRED(pu)\ is\_in\ msgs \\ \quad \Rightarrow next\_status(s, msgs, pu) = OK\end{array}$$

This means we must fix the STATUS_EVOLUTION specification and replace the above axiom by:

$$\begin{array}{l}\bullet\ status(s, pu) = FailureWithAck \wedge REPAIRED(pu)\ is\_in\ msgs \\ \quad \Rightarrow next\_status(s, msgs, pu) = OK\ when\ PU\_OK(s, msgs, pu) \\ \qquad\qquad\qquad\qquad\qquad\qquad else\ FailureWithoutAck\end{array}$$

Once the specification STATUS_EVOLUTION is modified as explained above, we can prove that the expected property holds.

To conclude, the reader should keep in mind that the validation of the specification is a very important task that deserves some serious attention. In this section we have only briefly illustrated some typical proof attempts that would naturally arise when validating the STEAM_BOILER_CONTROL_SYSTEM specification, and obviously many other proof attempts are required to reach a stage where we can trust our requirements specification of the steam-boiler control system.

C.10 Designing the Architecture of the Steam-Boiler Control System

We now have a validated requirements specification STEAM_BOILER_CONTROL_SYSTEM of the steam-boiler control system. The next step is to refine it into an architectural specification, thereby prescribing the intended architecture of the implementation of the steam-boiler control system. Indeed, the explanations given in Sec. C.3 suggest the following rather obvious architecture for the steam-boiler control system:

arch spec ARCH_SBCS =
units P : VALUE → PRELIMINARY;
S : PRELIMINARY → SBCS_STATE;
A : SBCS_STATE → SBCS_ANALYSIS;
C : SBCS_ANALYSIS → STEAM_BOILER_CONTROL_SYSTEM
result λV : VALUE • $C\,[A\,[S\,[P\,[V]]]]$
end

Note that we decide to describe the implementation of the steam-boiler control system as an open system, relying on an external component V implementing VALUE. This is consistent with our explanations in Sec. C.2: choosing a specific implementation of VALUE is obviously orthogonal to designing the implementation of the steam-boiler control system. This means in particular that the component V implementing VALUE will encapsulate the chosen representation of natural numbers and values, together with operations and predicates operating on them.

In a next step, we can refine the specification VALUE → PRELIMINARY of the component P into the following architectural specification:

arch spec ARCH_PRELIMINARY =
units SET : **{ sort** *Elem***}** × NAT → FINSET [**sort** *Elem*];
B : BASICS;
MS : MESSAGES_SENT **given** B;
MR : VALUE → MESSAGES_RECEIVED **given** B;
CST : VALUE → SBCS_CST
result λV : VALUE • $SET\,[MS$ **fit** *Elem* ↦ *S_Message*$]\,[V]$
and $SET\,[MR\,[V]$ **fit** *Elem* ↦ *R_Message*$]\,[V]$
and $CST\,[V]$
end

Here we decide to implement (generic) sets in a component *SET*, reused both for sets of messages received and sets of messages sent. Since the implementation of natural numbers is provided by the (external) component V, we use V for the second argument of the generic component *SET* in the result unit term.

The specification of the components C and S of ARCH_SBCS are simple enough that they do not need to be further architecturally refined. However, they are still not design specifications, and hence the development is not finished yet. For example, the specification of the component S (which implement the states of the steam-boiler control system) can be refined into the following specification, which provides a concrete implementation of states as a record of all the observable values.

```
from BASIC/STRUCTUREDDATATYPES get FINMAP

spec SBCS_STATE_IMPL =
     PRELIMINARY
then free type Status ::= OK | FailureWithoutAck | FailureWithAck
     free type ExtendedPumpState ::= sort PumpState | Unknown_PS
     free type
          ExtendedPumpControllerState ::= sort PumpControllerState
                                        |   SoonFlow | Unknown_PCS
     free type Valpair ::= pair(low : Value; high : Value)
then FINMAP [BASICS fit S ↦ PhysicalUnit] [sort Status]
and  FINMAP [BASICS fit S ↦ PumpNumber] [sort ExtendedPumpState]
and  FINMAP [BASICS fit S ↦ PumpNumber]
            [sort ExtendedPumpControllerState]
then free type ConcreteState ::=
          state(mode : Mode;
                nbSTOP : Nat;
                status_map : TotalFinMap[PhysicalUnit, Status];
                PS_predicted_map : TotalFinMap[PumpNumber,
                                               ExtendedPumpState];
                PCS_predicted_map : TotalFinMap[PumpNumber,
                                                ExtendedPumpControllerState];
                steam_predicted, level_predicted : Valpair)
     ops  status(s : ConcreteState; pu : PhysicalUnit) : Status
               = eval(pu, status_map(s));
          PS_predicted(s : ConcreteState; pn : PumpNumber)
               : ExtendedPumpState = eval(pn, PS_predicted_map(s));
          PCS_predicted(s : ConcreteState; pn : PumpNumber)
               : ExtendedPumpControllerState
               = eval(pn, PCS_predicted_map(s))
end
```

We need total finite maps here. The subsort *TotalFinMap* should be added to the finite map specification in the basic libraries. T.M.

The specification SBCS_STATE → SBCS_ANALYSIS of the component A of ARCH_SBCS can be refined into the following architectural specification:

arch spec ARCH_ANALYSIS =
units FD : SBCS_STATE → FAILURE_DETECTION;
PR : FAILURE_DETECTION → PU_PREDICTION;
ME : PU_PREDICTION → MODE_EVOLUTION [PU_PREDICTION];
MTS : MODE_EVOLUTION [PU_PREDICTION] → SBCS_ANALYSIS
result λS : SBCS_STATE • $MTS\,[ME\,[PR\,[FD\,[S]]]]$
end

In the above architectural specification ARCH_ANALYSIS, the component FD provides an implementation of failure detection, the component PR an implementation of the predicted state variables for the next cycle, the component ME provides an implementation of *next_mode* (and of *next_nbSTOP*), and the component MTS provides an implementation of *messages_to_send*.

The specifications of the components ME and MTS are simple enough to be directly implemented. The specifications of the components FD and PR can be refined as follows.

arch spec ARCH_FAILURE_DETECTION =
units $MTSF$: SBCS_STATE → MESSAGE_TRANSMISSION_SYSTEM_FAILURE;
PF : SBCS_STATE → PUMP_FAILURE;
PCF : SBCS_STATE → PUMP_CONTROLLER_FAILURE;
SF : SBCS_STATE → STEAM_FAILURE;
LF : SBCS_STATE → LEVEL_FAILURE;
PU : MESSAGE_TRANSMISSION_SYSTEM_FAILURE
× PUMP_FAILURE × PUMP_CONTROLLER_FAILURE
× STEAM_FAILURE × LEVEL_FAILURE
→ FAILURE_DETECTION
result λS : SBCS_STATE •
$PU\ [MTSF[S]]\ [PF[S]]\ [PCF[S]]\ [SF[S]]\ [LF[S]]$
hide *Pump_OK*, *Pump_Controller_OK*, *Steam_OK*, *Level_OK*
end

The above architectural specification ARCH_FAILURE_DETECTION refines the specification SBCS_STATE → FAILURE_DETECTION of the component FD in ARCH_ANALYSIS and introduces a component for each kind of failure detection. Then the component PU implements PU_OK, and in the result unit expression we hide the auxiliary predicates provided by the components PF, PCF, SF, and LF.[9]

[9] These auxiliary predicates are already hidden in the specification FAILURE_DETECTION. However, remember that in the specification of a generic component, the target specification is always an implicit extension of the argument specifications. This is why it is needed to hide the auxiliary predicates at the level of the result unit expression.

Finally we refine the specification FAILURE_DETECTION → PU_PREDICTION of the component *PR* of the architectural specification ARCH_ANALYSIS as follows:

arch spec ARCH_PREDICTION =
units *SE* : FAILURE_DETECTION →
STATUS_EVOLUTION [FAILURE_DETECTION];
SLP : FAILURE_DETECTION → STEAM_AND_LEVEL_PREDICTION;
PP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_STATE_PREDICTION;
PCP : STATUS_EVOLUTION [FAILURE_DETECTION]
× STEAM_AND_LEVEL_PREDICTION
→ PUMP_CONTROLLER_STATE_PREDICTION
result *λFD* : FAILURE_DETECTION •
local *SEFD* = *SE* [*FD*]; *SLPFD* = *SLP* [*FD*] **within**
PP [*SEFD*] [*SLPFD*] **and** *PCP* [*SEFD*] [*SLPFD*]
end

In the above architectural specification, the component *SE* provides an implementation of *next_status*. The component *SLP* provides an implementation of *next_steam_predicted*, *next_level_predicted*, *chosen_pumps*, and *Dangerous-WaterLevel*. The component *PP* provides an implementation of *next_PS_predicted*, and the component *PCP* provides an implementation of *next_PCS_predicted*.

We are now left with specifications of components that are simple enough to be directly implemented, and this concludes our case study.

C.11 Conclusion

Some concluding words will eventually be added here.

C.12 The Steam-Boiler Control Specification Problem

For completeness, the text describing the case study description, as originally provided by Jean-Raymond Abrial, is reproduced here.

C.12.1 Introduction

This text constitutes an informal specification of a program which serves to control the level of water in a steam-boiler. It is important that the program works correctly because the quantity of water present when the steam-boiler is working has to be neither too low nor to high; otherwise the steam-boiler or the turbine sitting in front of it might be seriously affected.

The proposed specification is derived fom an original text that has been written by LtCol. J.C. Bauer for the Institute for Risk Research of the University of Waterloo, Ontario, Canada. The original text has been submitted as a competition problem to be solved by the participants to the International Software Safety Symposium organized by the Institute for Risk Research. It has been given to us by the Institut de Protection et de Sûreté Nucléaire, Fontenay-aux-Roses, France. We would like to thank the author, the Institute for Risk Research, and the Institut de Protection et de Sûreté Nucléaire for their kind permission to use their text.

The text to follow is severely biased to a particular implementation. This is very often the case with industrial specifications that are rarely independent from a certain implementation people have in mind. In that sense, this specification is realistic. Your first formal specification steps could be *much more abstract* if that seems important to you (in particular if your formalism allows you to do so). In other words, you are encouraged to *structure* your specification in a way that is not necessarily the same as the one proposed in what follows. But in any case, you are asked to demonstrate that your specification can be refined to an implementation that is close enough to the functional requirements of the "specification" proposed below.

You might also judge that the specification contain some loose ends and inconsistencies. Do not hesitate to point them out and to take yourself some appropriate decisions. The idea, however, is that such inconsistencies should be solely within the *organization* of the system and *not within its physical properties*.

We are aware of the fact that the text to follow does not propose any precise model of the physical evolution of the system, only elementary suggestions. As a consequence, you may have to take some simple, even simplistic, abstract decisions concerning such a physical model.

C.12.2 Physical environment

The system comprises the following units:

- the steam-boiler,
- a device to measure the quantity of water in the steam-boiler,
- four pumps to provide the steam-boiler with water,
- four devices to supervise the pumps (one controller for each pump),
- a device to measure the quantity of steam which comes out of the steam-boiler,
- an operator desk,
- a message transmision system.

C.12.2.1 The steam-boiler

The steam-boiler is characterized by the following elements:

- A valve for evacuation of water. It serves only to empty the steam-boiler in its initial phase.
- Its total capacity C (indicated in litres).
- The minimal limit quantity M_1 of water (in litres). Below M_1 the steam-boiler would be in danger after five seconds, if the steam continued to come out at its maximum quantity without supply of water from the pumps.
- The maximal limit quantity M_2 of waters (in litres). Above M_2 the steam-boiler would be in danger after five seconds, if the pumps continued to supply the steam-boiler with water without possibility to evacuate the steam.
- The minimal normal quantity N_1 of water in litres to be maintained in the steam-boiler during regular operation ($M_1 < N_1$).
- The maximal normal quantity N_2 of water (in litres) to be maintained in the steam-boiler during regular operation ($N_2 < M_2$).
- The maximum quantity W of steam (in litres/sec) at the exit of the steam-boiler.
- The maximum gradient U_1 of increase of the quantity of steam (in litres/sec/sec).
- The maximum gradient U_2 of decrease of the quantity of steam (in litres/sec/sec).

C.12.2.2 The water level measurement device

The device to measure the level of water in the steam-boiler provides the following information:

- the quantity q (in litres) of water in the steam-boiler.

C.12.2.3 The pumps

Each pump is characterized by the following elements:

- Its capacity P (in litres/sec).
- Its functioning mode: on or off.
- It's being started: after having been switched on, the pump needs five seconds to start pouring water into the boiler (this is due to the fact that the pump does not balance instantaneously the pressure of the steam-boiler).
- It's being stopped: with instantaneous effect.

C.12.2.4 The pump control devices

Each pump controller provides the following information:

- the water circulates from the pump to the steam-boiler or, in the contrary, it does not circulate.

C.12.2.5 The steam measurement device

The device to measure the quantity of steam which comes out of the steam-boiler provides the following information:

- a quantity of steam v (in litres/sec).

C.12.2.6 Summary of constants and variables

The following table summarizes the various constants or physical variables of the system:

	Unit	**Comment**
	Quantity of water in the steam-boiler	
C	litre	Maximal capacity
M_1	litre	Minimal limit
M_2	litre	Maximal limit
N_1	litre	Minimal normal
N_2	litre	Maximal normal
	Outcome of steam at the exit of the steam-boiler	
W	litre/sec	Maximal quantity
U_1	litre/sec/sec	Maximum gradient of increase
U_2	litre/sec/sec	Maximum gradient of decrease
	Capacity of each pump	
P	litre/sec	Nominal capacity
	Current measures	
q	litre	Quantity of water in the steam-boiler
p	litre/sec	Throughput of the pumps
v	litre/sec	Quantity of steam exiting the steam-boiler

C.12.3 The overall operation of the program

The program communicates with the physical units through messages which are transmitted over a number of dedicated lines connecting each physical unit with the control unit. In first approximation, the time for transmission can be neglected.

The program follows a cycle and a priori does not terminate. This cycle takes place each five seconds and consists of the following actions:

- Reception of messages coming from the physical units.
- Analysis of informations which have been received.
- Transmission of messages to the physical units.

To simplify matters, and in first approximation, all messages coming from (or going to) the physical units are supposed to be received (emitted) *simultaneously* by the program at each cycle.

C.12.4 Operation modes of the program

The program operates in different modes, namely *initialization, normal, degraded, rescue, emergency stop.*

C.12.4.1 *Initialization* mode

The *initialization* mode is the mode to start with. The program enters a state in which it waits for the message STEAM-BOILER_WAITING to come from the physical units. As soon as this message has been received the program checks whether the quantity of steam coming out of the steam-boiler is really zero. If the unit for detection of the level of steam is defective (that is, when v is not equal to zero), the program enters the *emergency stop* mode. If the quantity of water in the steam-boiler is above N_2 the program activates the valve of the steam-boiler in order to empty it. If the quantity of water in the steam-boiler is below N_1 then the program activates a pump to fill the steam-boiler. If the program realizes a failure of the water level detection unit it enters the *emergency stop* mode. As soon as a level of water between N_1 and N_2 has been reached the program send continuously the signal PROGRAM_READY to the physical units until it receives the signal PHYSICAL_UNITS_READY which must necessarily be emitted by the physical units. As soon as this signal has been received, the program enters either the mode *normal* if all the physical units operate correctly or the mode *degraded* if any physical unit is defective. A transmission failure puts the program into the mode *emergency stop.*

C.12.4.2 *Normal* mode

The normal mode is the standard operating mode in which the program tries to maintain the water level in the steam-boiler between N_1 and N_2 with all physical units operating correctly. As soon as the water level is below N_1 or above N_2 the level can be adjusted by the program by switching the pumps on or off. The corresponding decision is taken on the basis of the information which has been received from the physical units. As soon as the program recognizes a failure of the water level measuring unit it goes into *rescue* mode. Failure of any other physical unit puts the program into *degraded* mode. If the water level is risking to reach one of the limit values M_1 or M_2 the

program enters the mode *emergency stop*. This risk is evaluated on the basis of a maximal behaviour of the physical units. A transmission failure puts the program into *emergency stop* mode.

C.12.4.3 *Degraded* mode

The *degraded* mode is the mode in which the program tries to maintain a satisfactory water level despite of the presence of failure of some physical unit. It is assumed however that the water level measuring unit in the steam-boiler is working correctly. The functionality is the same as in the preceding case. Once all the units which were defective have been repaired, the program comes back to *normal* mode. As soon as the program sees that the water level measuring unit has a failure, the program goes into mode *rescue*. If the water level is risking to reach one of the limit values M_1 or M_2 the program enters the mode *emergency stop*. A transmission failure puts the program into *emergency stop* mode.

C.12.4.4 *Rescue* mode

The *rescue* mode is the mode in which the program tries to maintain a satisfactory water level despite of the failure of the water level measuring unit. The water level is then estimated by a computation which is done taking into account the maximum dynamics of the quantity of steam coming out of the steam-boiler. For the sake of simplicity, this calculation can suppose that exactly n liters of water, supplied by the pumps, do account for exactly the same amount of boiler contents (no thermal expansion). This calculation can however be done only if the unit which measures the quantity of steam is itself working and if one can rely upon the information which comes from the units for controlling the pumps. As soon as the water measuring unit is repaired, the program returns into mode *degraded* or into mode *normal*. The program goes into *emergency stop* mode if it realizes that one of the following cases holds: the unit which measures the outcome of steam has a failure, or the units which control the pumps have a failure, or the water level risks to reach one of the two limit values. A transmission failure puts the program into *emergency stop* mode.

C.12.4.5 *Emergency stop* mode

The *emergency stop* mode is the mode into which the program has to go, as we have seen already, when either the vital units have a failure or when the water level risks to reach one of its two limit values. This mode can also be reached after detection of an erroneous transmission between the program and the physical units. This mode can also be set directly from outside. Once the program has reached the *Emergency stop* mode, the physical environment is then responsible to take appropriate actions, and the program stops.

C.12.5 Messages sent by the program

The following messages can be sent by the program:

- MODE(m): The program sends, at each cycle, its current mode of operation to the physical units.
- PROGRAM_READY: In *initialization* mode, as soon as the program assumes to be ready, this message is continuously sent until the message PHYSICAL_UNITS_READY coming from the physical units has been received.
- VALVE: In *initialization* mode this message is sent to the physical units to request opening and then closure of the valve for evacuation of water from the steam-boiler.
- OPEN_PUMP(n): This message is sent to the physical units to activate a pump.
- CLOSE_PUMP(n): This message is sent to the physical units to stop a pump.
- PUMP_FAILURE_DETECTION(n): This message is sent (until receipt of the corresponding acknowledgement) to indicate to the physical units that the program has detected a pump failure.
- PUMP_CONTROL_FAILURE_DETECTION(n): This message is sent (until receipt of the corresponding acknowledgement) to indicate to the physical units that the program has detected a failure of the physical unit which controls a pump.
- LEVEL_FAILURE_DETECTION: This message is sent (until receipt of the corresponding acknowledgement) to indicate to the physical units that the program has detected a failure of the water level measuring unit.
- STEAM_FAILURE_DETECTION: This message is sent (until receipt of the corresponding acknowledgement) to indicate to the physical units that the program has detected a failure of the physical unit which measures the outcome of steam.
- PUMP_REPAIRED_ACKNOWLEDGEMENT(n): This message is sent by the program to acknowledge a message coming from the physical units and indicating that the corresponding pump has been repaired.
- PUMP_CONTROL_REPAIRED_ACKNOWLEDGEMENT(n): This message is sent by the program to acknowledge a message coming from the physical units and indicating that the corresponding physical control unit has been repaired.
- LEVEL_REPAIRED_ACKNOWLEDGEMENT: This message is sent by the program to acknowledge a message coming from the physical units and indicating that the water level measuring unit has been repaired.
- STEAM_REPAIRED_ACKNOWLEDGEMENT: This message is sent by the program to acknowledge a message coming from the physical units and indicating that the unit which measures the outcome of steam has been repaired.

C.12.6 Messages received by the program

The following messages can be received by the program:

- STOP: When the message has been received three times in a row by the program, the program must go into *emergency stop*.
- STEAM_BOILER_WAITING: When this message is received in *initialization* mode it triggers the effective start of the program.
- PHYSICAL_UNITS_READY: This message when received in *initialization* mode acknowledges the message PROGRAM_READY which has been sent previously by the program.
- PUMP_STATE(n, b): This message indicates the state of pump n (open or closed). This message must be present during each transmission.
- PUMP_CONTROL_STATE(n, b): This message gives the information which comes from the control unit of pump n (there is flow of water or there is no flow of water). This message must be present during each transmission.
- LEVEL(v): This message contains the information which comes from the water level measuring unit. This message must be present during each transmission.
- STEAM(v): This message contains the information which comes from the unit which measures the outcome of steam. This message must be present during each transmission.
- PUMP_REPAIRED(n): This message indicates that the corresponding pump has been repaired. It is sent by the physical units until a corresponding acknowledgement message has been sent by the program and received by the physical units.
- PUMP_CONTROL_REPAIRED(n): This message indicates that the corresponding control unit has been repaired. It is sent by the physical units until a corresponding acknowledgement message has been sent by the program and received by the physical units.
- LEVEL_REPAIRED: This message indicates that the water level measuring unit has been repaired. It is sent by the physical units until a corresponding acknowledgement message has been sent by the program and received by the physical units.
- STEAM_REPAIRED: This message indicates that the unit which measures the outcome of steam has been repaired. It is sent by the physical units until a corresponding acknowledgement message has been sent by the program and received by the physical units.
- PUMP_FAILURE_ACKNOWLEDGEMENT(n): By this message the physical units acknowledge the receipt of the corresponding failure detection message which has been emitted previously by the program.
- PUMP_CONTROL_FAILURE_ACKNOWLEDGEMENT(n): By this message the physical units acknowledge the receipt of the corresponding failure detection message which has been emitted previously by the program.

- LEVEL_FAILURE_ACKNOWLEDGEMENT: By this message the physical units acknowledge the receipt of the corresponding failure detection message which has been emitted previously by the program.
- STEAM_FAILURE_ACKNOWLEDGEMENT: By this message the physical units acknowledge the receipt of the corresponding failure detection message which has been emitted previously by the program.

C.12.7 Detection of equipment failures

The following erroneous kinds of behaviour are distinguished to decide whether certain physical units have a failure:

- PUMP: (1) Assume that the program has sent a start or stop message to a pump. The program detects that during the following transmission that pump does not indicate its having effectively been started or stopped. (2) The program detects that the pump changes its state spontaneously.
- PUMP_CONTROLLER: (1) Assume that the program has sent a start or stop message to a pump. The program detects that during the second transmission after the start or stop message the pump does not indicate that the water is flowing or is not flowing; this despite of the fact that the program knows from elsewhere that the pump is working correctly. (2) The program detects that the unit changes its state spontaneously.
- WATER_LEVEL_MEASURING_UNIT: (1) The program detects that the unit indicates a value which is out of the valid static limits (that is, between 0 and C). (2) The program detects that the unit indicates a value which is incompatible with the dynamics of the system.
- STEAM_LEVEL_MEASURING_UNIT: (1) The program detects that the unit indicates a value which is out of the valid static limits (that is, between 0 and W). (2) The program detects that the unit indicates a value which is incompatible with the dynamics of the system.
- TRANSMISSION: (1) The program receives a message whose presence is aberrant. (2) The program does not receive a message whose presence is indispensable.

References

The bibliography style is to be numerical in the final version.

AM02. S. Autexier and T. Mossakowski. Integrating HOL-CASL into the development graph manager MAYA. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop*, volume 2309 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2002.

BDH+01. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a component-based language development environment. In R. Wilhelm, editor, *CC'01*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

BJKO00. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice & Experience*, 30:259–291, 2000.

BK98. J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.

BMV03. M.G.J. van den Brand, P.E. Moreau, and J.J. Vinju. Environments for term rewriting engines for free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

BSVV02. M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for Scannerless Generalized LR parsers. In R. Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *LNCS*, pages 143–158. Springer-Verlag, 2002.

CoF. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

CRV03. Feng Chen, Grigore Rosu, and Ram Prasad Venkatesan. Rule-based analysis of dimensional safety. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 197–207. Springer-Verlag, 2003.

DHK96. A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific, 1996.

GB92. Joseph A. Goguen and Rodney M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

KR00. Hélène Kirchner and Christophe Ringeissen. Executing CASL equational specifications with the ELAN rewrite engine. Note T-9 (revised version), in [CoF], November 2000.

Mos00. Till Mossakowski. CASL: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 93–108. Springer-Verlag, 2000.

Mos02. Till Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.

Mos03. Peter D. Mosses, editor. CASL *Reference Manual*. Lecture Notes in Computer Science. Springer, 2003. To appear.

Pau94. L. C. Paulson. *Isabelle – A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994.

List of Named Specifications

Index of Library and Specification Names