LCM 3.0
A Language for Describing Conceptual Models
Syntax Definition[1]

Remco Feenstra[2] Roel Wieringa
Faculty of Mathematics and Computer Science
Vrije Universiteit
De Boelelaan 1081a
1081 HV Amsterdam
email rbfeens|roelw@cs.vu.nl

December 21, 1993

[1]This research is partially supported by Esprit Basic Research Action IS–CORE (working group 6071).
[2]Supported by the Foundation for Computer Science in the Netherlands (SION) with financial support form the Netherlands Organization for Scientific Research (NWO), project 612–317–408.

Contents

Chapter 1

Preliminaries

LCM (Language for Conceptual Modeling) is a specification language with which conceptual models (CM's) can be specified. We assume that the CM's specified in LCM are used to represent some part of reality that is to be represented by a database system (DBS). A subset of LCM is intended to be executable.

Version 3.0 of the LCM specification language is a revision of version 2 of the language, formerly known as CMSL (Conceptual Model Specification Language)[12], which in turn is based on the language described in Wieringa's Ph.D. thesis [11].

Experience with different styles of writing CMSL specifications suggested many improvements and extensions to the language. This report describes a revised syntax of the language based on these (often incompatible) suggestions, while at the same time paying attention to requirements stemming from the intention to provide computer support for manipulating specifications. These include restriction of the notation to the ASCII character set, the standardization of names of built-in functions and many other notational issues.

This revision process included changing the name of the language from CMSL to LCM, because the abbreviation CMSL proved to be unpronouncable to most people. Besides, it suggests its relation to MCM (Method for Conceptual Modeling), which is described in a companion report[13]. A second companion report describes the application of MCM to a case study, in which a LCM specification is developed[14].

This document intends to serve as a reference for the specifier in the area of syntactic issues of LCM 3.0.

Although the text and examples in this document suggest informally a meaning of the specifications, the formal semantics of LCM 3.0 is not given here. The interested reader should consult [11] for more on a formal semantics of LCM.

1.1 Terminology

We start by fixing some terminology that will be used throughout this report.

- By an **object** we mean a communicating system. That is, an object is an observable part of the world with a state space, state transitions, a life cycle, and communications with other objects. The life cycle of an object o is composed of events, and each event may be a communication with other objects and may cause a state transition in the life of o. Each state transition of o is caused by an event in the life of o.

- The **subject** of an event is the object in whose life the event occurs.

This concept of object is liberal and also includes relationships between objects. So if we wish, we can regard the employment relationship between a person and his or her employer as itself an object, with a state space, state transitions, etc.

A basic distinction in conceptual modeling is that between a *database system* (DBS) and the *universe of discourse* (UoD) of the database system. A DBS is a system that can register data. The UoD is that part of the world about which the DBS registers data. A message from the UoD to the DBS that an event has occurred in the UoD is called a *registration event*.

Objects may exist in the UoD or in the DBS. Examples of UoD objects are persons, chairs, factories and students, examples of DBS objects are windows, records, cursors, and files. (We may of course maintain a DBS about objects in another DBS. The second DBS is then the UoD of the first DBS.)

- An **identifier** is a proper name for an object such that the **designation** relation between the name and the object is 1–1 and monotonically growing. This means tant

  1. that each identifier names at most on object out of a set of possible objects called an **objectspace**,
  2. that each object is named by at most on identifier from a set of possible identifiers called a **namespace**, and
  3. that the designation relation between an identifier and an object, once it exists, never changes.

  The first two requirements are also called the **singularity requirements**.

For each identifier, we must indicate its objectspace, its namespace and the state space in which the identifier is unchanging. For example, the employee number in use in a company is an identifier with all employees of the company as objectspace, all strings with a certain structure as namespace, and the set of all possible states of the identified employees as state space.

It is clear that the concept of an identifier is an idealization. Fraud with employee numbers, children and dependants who are identified by the employee number of another person, errors in assigning employee numbers, reorganizations of the namespace of employee numbers, etc. are examples of cases where at least one of the three identifier conditions is violated. These violations can occur in most identifiers, such as social security numbers, student identification numbers, etc. Problems with object identification are discussed in detail by Wieringa and de Jonge [15].

Each identifier must be *generated* so as to guarantee its uniqueness in a namespace, and *assigned* to an object in order to guarantee singularity with respect to an objectspace. Finally, *rigidness* must be guarded, enforced, guaranteed, etc. These tasks may be done by one or more persons, organizations or machines.

- An **internal identifier** is an identifier generated by a DBS and assigned to internal objects. The three identifier requirements are usually guaranteed by the DBS.

- An **external identifier** is an identifier assigned to external objects, whose identifier requirements are guaranteed by persons, organizations or machines other than the DBS. External identifiers are not usually generated by a DBS.

Examples of an internal identifier is a unique identification of database records, unique process identifications, etc. Examples of external identifiers are student numbers, social security numbers, employee numbers etc. The important property of internal identifiers, as far as the DBS is concerned, is that their generation and assignment to objects is done by the DBS itself and that their monotonic designation is enforced by the DBS itself. This means that the namespace, objectspace and state space of internal identifiers are known to the DBS itself. Again, the concept of an internal identifier is an idealization. For example, namespaces may be finite and this causes names to be reused as identifiers for different internal objects. In LCM, however, we use the ideal concept of internal identifiers.

We also allow the definition of external identifiers, but the identity requirements for external identifiers cannot be specified in a DBS specification, because the namespace, objectspace and state space may not be known to the DBS. For example, there may be more social security numbers

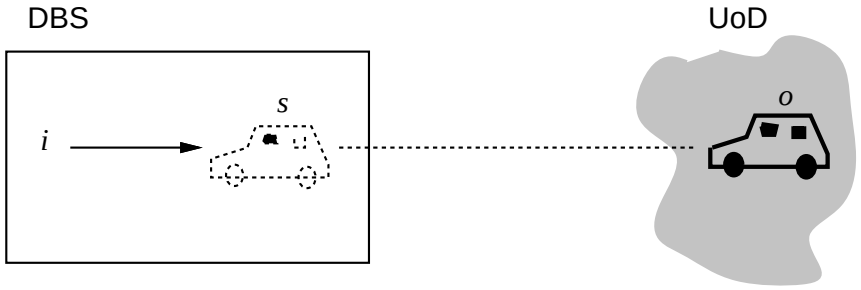

Figure 1.1: The internal identifier i is assigned to the internal object s, which is used to represent a car o in the UoD.

than those known to the DBS, there may be more persons with social security numbers than those known to the DBS, and the assignment of social security numbers to persons may change without the DBS being able to prevent it. The use of external identifiers is supported in LCM by allowing an object attribute to be declared unique and fixed. To repeat, this does not guarantee that the attribute is an external identifier.

There are two identifier-related concepts in databases, that of key and surrogates.

- A **key** is a combination of attributes that is unique in every possible state of the world.
- A **surrogate** is an identifier generated by a DBS with an external objectspace.

The combination of name, birthplace, birthdate has been used as key for persons, and employee numbers are often treated as keys rather than as identifiers. It is possible that a key of an object changes, so that monotonic designation is violated and one object may have two keys in its life. It is also possible that two objects, that exist at different times, have the same key, so that singularity is violated. LCM provides a facility to define keys by allowing a combination of object attributes to be declared unique in every state of the world.

An example of a surrogate is a userid generated by a DBS and used to identify users of a DBS. (One person may be different users.) Surrogates are usually required to be invisible for users. LCM has no facilities to support the definition of surrogates by users. However, we will use internal identifiers as surrogates. This is done as follows.

Each UoD object is represented in the DBS by an internal DBS object. For example, a car in the UoD is represented in the DBS by an internal object o, that has a state space, state transitions, a life cycle and a communicating behavior that mirrors that of the represented car. At any moment, this object exists as a part of the state of a computer. For each DBS object, an internal identifier s is generated by the DBS and assigned to the internal object by the DBS. This is illustrated in figure 1.1.

The DBS will use this internal identifier as reference to the internal object o. The domain of this identifier is the set of all internal DBS objects, the namespace of the identifier is the set of all possible internal identifiers. This means that the internal identifier s is used as a *surrogate* for the real-world car.

- A **class** is the set of all possible objects, also called **class instances**, that share a number of properties.
- The **existence set** of a class in a state of the UoD is the set of existing objects of that class in that state of the UoD.

Examples of classes are *PERSON*, *STUDENT*, *CAR* etc. In each state of the UoD, a subset of the set of all possible persons exist and this is the existence set of *PERSON*.

- A **value** is an abstract, unobservable entity. Values are often called data.

- A **value sort** is the set of all possible values that share a number of properties. Value sorts are often called data types.

Examples of value sorts are the set of all integers, the set of all strings, etc. We regard characters and strings as values, i.e. abstract and unobservable entities. However, characters and strings may have concrete, observable *notations*, just as integers and rational numbers have concrete, visible notations.

- The **type** of a set of entities is the set of all properties shared by the entities in the set. If the entities are objects, we speak of **object type**, if they are values, we speak of **value type**.

For each class, we can define its corresponding type, i.e. the set of all properties shared by all possible instances of the class. Since one object may be an element of several classes at the same moment, one object may have several types. A similar observation is applicable to values.

For each class C, there is a value sort of the same namethat contains all internal identifiers for objects of that class. As pointed out above, these internal identifiers are surrogates for real-world objects.

1.2 Extended Backus-Naur Form (EBNF)

In order to define the syntax of the LCM specification language we will use natural language combined with Extended Backus-Naur Form (EBNF) expressions. The latter is used in two different contexts, with a slightly different meaning. First, we use it in chapter 2 to define the lexical grammar, which deals with the appearance of items such as numbers and names, and describe how these are built from single characters. In this case white space in the EBNF rules is significant. Second, we use it in the other chapters to describe the context free grammar of LCM, using the top-level nonterminals of the lexical definitions as terminal symbols of the context-free grammar. The nonterminals in each of the specifications are written between a "⟨" and a "⟩", such as ⟨Name⟩ and ⟨LCMSpecification⟩, or as themselves in typewriter font in the case of symbols and keywords, e.g. `axioms` and `&`. For convenience, EBNF uses the following constructs in its rules:

- ⟨NonTerminal⟩* means zero or more occurrences of ⟨NonTerminal⟩,
- ⟨NonTerminal⟩+ means one or more occurrences of ⟨NonTerminal⟩,
- { ⟨NonTerminal⟩ ⟨separator⟩ }+ means one or more occurrences of ⟨NonTerminal⟩, separated by ⟨separator⟩s between them,
- *[* ⟨NonTerminal⟩ *]* means zero or one occurrence of ⟨NonTerminal⟩,
- ⟨NonTerminal1⟩ | ⟨NonTerminal2⟩ means ⟨NonTerminal1⟩ or ⟨NonTerminal2⟩.

Please note the subtle difference between "*[*" and "[" appearing in EBNF definitions: the first symbol is part of EBNF and indicates zero or one occurrences, whereas the second is the literal square bracket character that represents itself.

The start symbol of the context free grammar of LCM is ⟨LCM-Specification⟩.

Chapter 2

Lexical syntax

The language described here is designed to be usable on hardware that only supports ASCII input and output facilities.

In this chapter, we give a grammar that defines the appearance of syntactic entities that occur als terminals in the context-free grammar of the next section. The reserved words and predefined names appear as themselves in the context-free grammar. The symbols ⟨Name⟩, ⟨ZeroConstant⟩, ⟨RationalConstant⟩, ⟨PosIntConstant⟩, ⟨CharacterConstant⟩ and ⟨StringConstant⟩ are nonterminals in the grammar of this chapter. In the grammars in the next chapters, however, they are treated as additional terminal symbols.

2.1 Lexical elements

A LCM specification consists of a (possibly empty) sequence of various kinds of **lexical elements**. The boundaries between the lexical elements occurring in the specification are determined by the next EBNF definition, together with the following

> **Lexical item identification rule** The next lexical item starting at some position (if any) is the longest possible substring of the specification text starting at that position that constitutes a valid ⟨LexicalElement⟩.

⟨LCM-SpecificationText⟩ —→ ⟨LexicalElement⟩*

⟨LexicalElement⟩ —→ ⟨ReservedWord⟩ | ⟨Name⟩ | ⟨Constant⟩ | ⟨WhiteSpace⟩

LCM is case sensitive: upper case characters are considered to be different from their lower case counterparts. This applies not only to names, but also to reserved words.

In the sections below, we will discuss the structure of the various lexical elements.

Example

Consider the following specification text:

```
     begin
CLASS  classification
```

It is split into the following ⟨LexicalElement⟩s:

- a ⟨WhiteSpace⟩ for the 4 leading space characters,
- the keyword `begin`,

- another ⟨WhiteSpace⟩ (the newline), a ⟨Name⟩ (although `class` is a reserved word, `CLASS` isn't, because LCM is case-sensitive),
- a ⟨WhiteSpace⟩ and a ⟨Name⟩ (although `classification` starts with the same characters as the reserved word `class`, the substring `classification` is longer than the substring `class`, which is a reserved word),
- a final ⟨WhiteSpace⟩ for the newline.

2.2 White space

White space consists of the juxtaposition of one or more occurrences of white space elements:

⟨WhiteSpace⟩ ⟶ ⟨WhiteSpaceElement⟩+

The following characters each count as one white space element:

1. the space character (ASCII SP)
2. the carriage return character (ASCII CR)
3. the newline character (ASCII LF)
4. the horizontal tabulation character (ASCII HT)
5. the form feed character character (ASCII FF)
6. the vertical tab character (ASCII VT)
7. a comment (see 2.6).

White space has no significance except to separate ⟨Name⟩s, reserved words and ⟨Constant⟩s. It can be added freely by the author of a specification to increase readability, as long as the rules for partitioning the specification text into ⟨LexicalElements⟩ are kept in mind to avoid accidentally splitting up other ⟨LexicalElement⟩s.

Example

As an example how the inclusion of white space can influence the meaning of a specification by defining the boundaries between lexical elements, consider the difference between `Route66` and `Route 66`. The first is a ⟨Name⟩, whereas the latter is a ⟨Name⟩ followed by ⟨WhiteSpace⟩ followed by a ⟨PosIntConstant⟩. Although the ⟨WhiteSpace⟩ is ignored in the context free grammar, the difference between one and two other ⟨LexicalElements⟩ should be clear.

2.3 Reserved words

The **reserved words** of LCM are:

⟨ReservedWord⟩ ⟶ `after` | `and` | `attributes` | `axioms` | `begin` | `bijection` | `by` | `class` | `components` | `creation` | `cycle` | `decompositions` | `deletion` | `div` | `empty` | `end` | `events` | `fail` | `fails` | `fixed` | `forall` | `function` | `functions` | `identifiers` | `if` | `In` | `initially` | `injection` | `inverse` | `keys` | `life` | `necessarily` | `not` | `object` | `of` | `only` | `or` | `partitioned` | `possibly` | `predicate` | `predicates` | `relationship` | `services` | `skip` | `specialization` | `specialized` | `succeeds` | `surjection` | `transactions` | `type` | `Value` | `;` | `,` | `::` | `(` | `)` | `[` | `]` | `->` | `<->` | `+` | `-` | `*` | `/` | `&` | `||` | `.` | `=` | `<` | `<=` | `>=` | `>` | `^` | `!=`

2.4 Names

A **name** can be used to refer to a construction in a LCM specification. A ⟨Name⟩ has the appearance as described below, with the additional constraint that a ⟨Name⟩ cannot consist of the same sequence of characters as a ⟨ReservedWord⟩.

⟨Name⟩ ⟶ ⟨NameStartCharacter⟩ ⟨NameContinuationCharacter⟩*

⟨NameStartCharacter⟩ ⟶ ⟨Letter⟩ | _

⟨NameContinuationCharacter⟩ ⟶ ⟨Letter⟩ | ⟨DecimalDigit⟩ | _

⟨Letter⟩ ⟶ a | b | c | ··· | z | A | B | C | ··· | Z

⟨DecimalDigit⟩ ⟶ 0 | ⟨NonZeroDecimalDigit⟩

⟨NonZeroDecimalDigit⟩ ⟶ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Some ⟨Name⟩s are predefined, and may not be redefined. These are the ⟨PredefinedSortName⟩s (see section 5.1), the built-in nullary and unary predicate names shown in tables 5.3 and 5.4 and the built-in prefix function names of table 5.8.

Names starting with a underscore (_) are reserved, and cannot be used by the authors of specifications.

Examples

Some examples of ⟨Name⟩s are: `MEMBER`, `date_borrowed`, `true` (the reserved word `True` is capitalized), `Married` and `Agent007`.

The following are not valid names: `#times_borrowed` (starts with a non-letter), `cycle`, `True` (they are reserved words) and `Fish&Chips` (contains other characters than letters, digits and underscore. `_fine` is a valid name, but since it starts with an underscore it is reserved.

See also section 3.4 on the scope of names.

2.5 Constants

2.5.1 Number constants

We distinguish three kinds of **number constants**: ⟨ZeroConstant⟩s, that represent the number zero, and ⟨PosIntConstant⟩s, that represent positive integers and ⟨RationalConstants⟩, that represent rational numbers in decimal notation.

⟨NumberConstant⟩ ⟶ ⟨ZeroConstant⟩ | ⟨PosIntConstant⟩ | ⟨RationalConstant⟩

⟨ZeroConstant⟩ ⟶ ⟨Zero⟩+

⟨Zero⟩ ⟶ 0

⟨PosIntConstant⟩ ⟶ ⟨Zero⟩+ ⟨NonZeroDecimalDigit⟩ ⟨DecimalDigit⟩*

⟨RationalConstant⟩ ⟶ ⟨DecimalDigit⟩+ . ⟨DecimalDigit⟩+

Examples

Some examples of ⟨NumberConstant⟩s are: The ⟨PosIntConstant⟩ `42`, the ⟨PosIntConstant⟩ `007`, the ⟨ZeroConstants⟩ `0` and `000000`, and the ⟨RationalConstant⟩s `1.0`, `3.14` and `0.0` (the latter is *not* a ⟨ZeroConstant⟩.

Note that leading signs are not part of a number constant; `-10` is an expression consisting of the unary negation operator applied to the constant `10` of sort `POSINT`, turning it into an expression of sort `NEGINT`.

2.5.2 Character constants

The exact set of **character constants** is implementation dependent, but it is required that it includes at least all printable characters in the ASCII set, with the following special conventions that may be used for the unprintable ones (or as alternatives for the printables).

⟨CharacterConstant⟩ ⟶ ' ⟨CharacterSpecification⟩ '

⟨CharacterSpecification⟩ ⟶ any single printable character | ⟨SpecialCharacterConstant⟩

⟨SpecialCharacterConstant⟩ ⟶ `\n` | `\r` | `\f` | `\t` | `\b` | `\"` | `\`⟨OctalDigits⟩

⟨OctalDigits⟩ ⟶ ⟨OctalDigit⟩ | ⟨OctalDigit⟩ ⟨OctalDigit⟩ | ⟨OctalDigit⟩ ⟨OctalDigit⟩ ⟨OctalDigit⟩

⟨OctalDigit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

The intended meaning of the ⟨SpecialCharacterConstant⟩s, which are notations for nonprintable characters are:

1. `\n` is the newline character (ASCII LF),
2. `\r` is the carriage return character (ASCII CR),
3. `\f` is the form feed character (ASCII FF),
4. `\t` is the tab character (ASCII VT),
5. `\b` is backspace character (ASCII BS),
6. `\"` is the double quote character (only needed inside ⟨StringConstant⟩s, see 2.5.3).
7. The octal digits after the backslash represent the character with ASCII code given by the ⟨Octal-digits⟩.

2.5.3 String constants

⟨StringConstant⟩ ⟶ " ⟨CharacterSpecification⟩* "

A string consists of a sequence of ⟨CharacterSpecification⟩s, which have the same form as in ⟨CharacterConstant⟩s. Null characters may appear as part of the string constants. To include a double quote character in a string constant, use the ⟨SpecialCharacterConstant⟩ `\"`, otherwise it will signal the end of the ⟨StringConstant⟩ (see also 2.5.2).

Examples

`"Basil Fawlty"`, `""` and `"\""` are all valid ⟨StringConstant⟩s, the second being an example of an empty string constant, whereas the third is an example of a string constant containing the double quote character.

2.6 Comments

A **comment** may be used to clarify the specification text to human readers, but has no meaning except that it separates ⟨LexicalElement⟩s. Its syntax is:

⟨Comment⟩ ⟶ `--` ⟨NonNewLineCharacter⟩*

As a consequence of this definition, comments cannot be nested.

Examples

The following five lines each contain exactly one ⟨Comment⟩:

```
--
-- This is a comment. Testing 1, 2, 3,...
-- Comments do -- not -- nest.
-- "this is --not-- a string constant."
---
```

As a consequence of the rule that the lexical item starting at some position is the longest possible substring of the specification text starting at that position that constitutes a valid ⟨LexicalElement⟩, two adjacent hyphens occuring inside a ⟨StringConstant⟩ like

```
"a string constant with ---- hyphens inside"
```

will *not* signal the start of a comment. Along the same lines, the text

```
--3
```

does not start with the ⟨ReservedWord⟩ `-`, but with the (longer) ⟨LexicalElement⟩ ⟨Comment⟩! If we really wanted to write the negation of the negation of three, we could have written `-(-3)` instead.

Chapter 3

Names, declarations and scope

In section 2.4 the lexical structure of a ⟨Name⟩ is defined. Each occurrence of a name in a LCM specification has a meaning, but in the presence of LCM's facilities for overloading names and scoping, the name alone often isn't sufficient to determine to what it refers; in order to do this, we need to take into account the position and the context in which it occurs in the LCM specification. In this chapter, we introduce some terminology to enable a concise description of the facilities for name overloading and hiding in the rest of this document. In addition, we formulate an important constraint that every valid LCM specification should satisfy, which says that all names that occur in the LCM specification should have a unique point in the specification where they are defined. The notions introduced below deal with defining which of the occurrences of names in a LCM specification have the same meaning. First, we need to distinguish names from occurrences of names.

3.1 Names and occurrences of a name

With a **name** (note that this is different from a ⟨Name⟩) we mean a particular string of characters that satisfies the lexical syntax of ⟨Name⟩ as given in subsection 2.4. Names may appear in LCM specifications, but this isn't necessary; we may talk about names irrespective of a particular specification, like about the name `Foo`. We say that two names are equal iff they consist of the same sequence of characters.

Every lexeme in a particular LCM specification which is a name will be called an **occurrence of a name** in that specification. All occurrences of a name in a specification are distinct. Each occurrence of a name in a specification has a unique **position** in that specification. In the sequel, we will assume that a particular LCM specification is understood, and talk simply of "occurrences of a name" instead of "occurrences of a name at some position in some LCM specification". An occurrence of a name is called **an occurrence of name** m, iff it consists of the same sequence of characters as the name m. A LCM specification may contain zero, one or many occurrences of a name m

We will say that two occurrences of a name are **occurrences of the same name** iff there is a name n such that both are occurrences of name n.

3.2 Properties of occurrences of names

We assume that the specification contains a set of name occurrences for which the concepts of *position* of a name occurrence and *region* in the specification are defined. For example, in a specification that is presented as a sequence of characters (e.g. a text file) a position of a name occurrence might be the number of characters that precede the first characters of the name, starting to count from the beginning of the sequence. A region might be defined as a subsequence

of the specification. In a hypertext-like system, the concepts of position and region will have other definitions.

With each occurrence q of a name we associate the following properties:

1. its position, notated as $Position(q)$, which is the place in the LCM specification where this occurrence is found.
2. its syntactical kind, notated as $Kind(q)$, with $Kind(q) \in \{\mathsf{sort}, \mathsf{predicate}, \mathsf{attribute}, \mathsf{function}, \mathsf{event}, \mathsf{transaction}, \mathsf{process}, \mathsf{relationship}, \mathsf{variable}, \mathsf{service}, , \mathsf{component}, \mathsf{object}\}$.

Dependent on the kind of an occurrence q, it can have other properties:

- If $Kind(q) \in \{\mathsf{predicate}, \mathsf{attribute}, \mathsf{function}, \mathsf{process}\}$, it has the following two properties
  - an arity, notated as $Arity(q)$, which is the number of arguments the occurrence takes. $Arity(q)$ may be zero.
  - a sequence $ParameterTypes(q)$ of length $Arity(q)$ of parameter type expressions.
- If $Kind(q) \in \{\mathsf{attribute}, \mathsf{function}, \mathsf{component}\}$, it has an additional property $SmallestResultSort(q)$.
- If $Kind(q) = \mathsf{sort}$, it has the following two additional properties
  - The number of arguments the sort takes, written as $SortArity(q)$. $SortArity(q)$ may be zero for simple sorts.
  - A sequence of length $SortArity(q)$ of occurrences of names that are the (formal or actual) parameters of the sort, written as $SortParameters(q)$.

The actual values of these properties are discussed in the text that accompanies the syntax rules in which the name may appear. Figure 3.1 illustrates the relevant concepts in a diagram.

3.3 Defining occurrences of a name

In LCM, due to overloading and local definitions hiding more global ones, two occurrences of the same name can mean different things. We need some means to express when two occurrences of a name mean the same thing. This is done by marking some special occurrences of a name in the LCM specification text as **defining occurrences of a name**. These can be thought of as being part of a declaration of that name. Which of the occurrences of a name are defining occurrences of that name is discussed with the syntax rules for the declaration that contains them.

In section 3.5 we will give rules that associate exactly one defining occurrence with each occurrence of a name at some position in a correct LCM specification, such that they are occurrences of the same name.

In this way we have implicitly defined which occurrences of a name refer to the same thing, viz. all occurrences that have the same defining occurrence.

3.4 Scope of a defining occurrence

In addition to the properties that are common to all occurrences of a name, a defining occurrence q also has a **scope**, written as $Scope(q)$. Intuitively, the scope of an occurrence of a name is a region in the LCM specification surrounding q such that the effects of the defining occurrence q are restricted to $Scope(q)$: if an occurrence of a name has q as a defining occurrence, then the position of that occurrence lies within $Scope(q)$.

The scope of a defining occurrence of a name will be discussed with the syntax rules containing that defining occurrence.

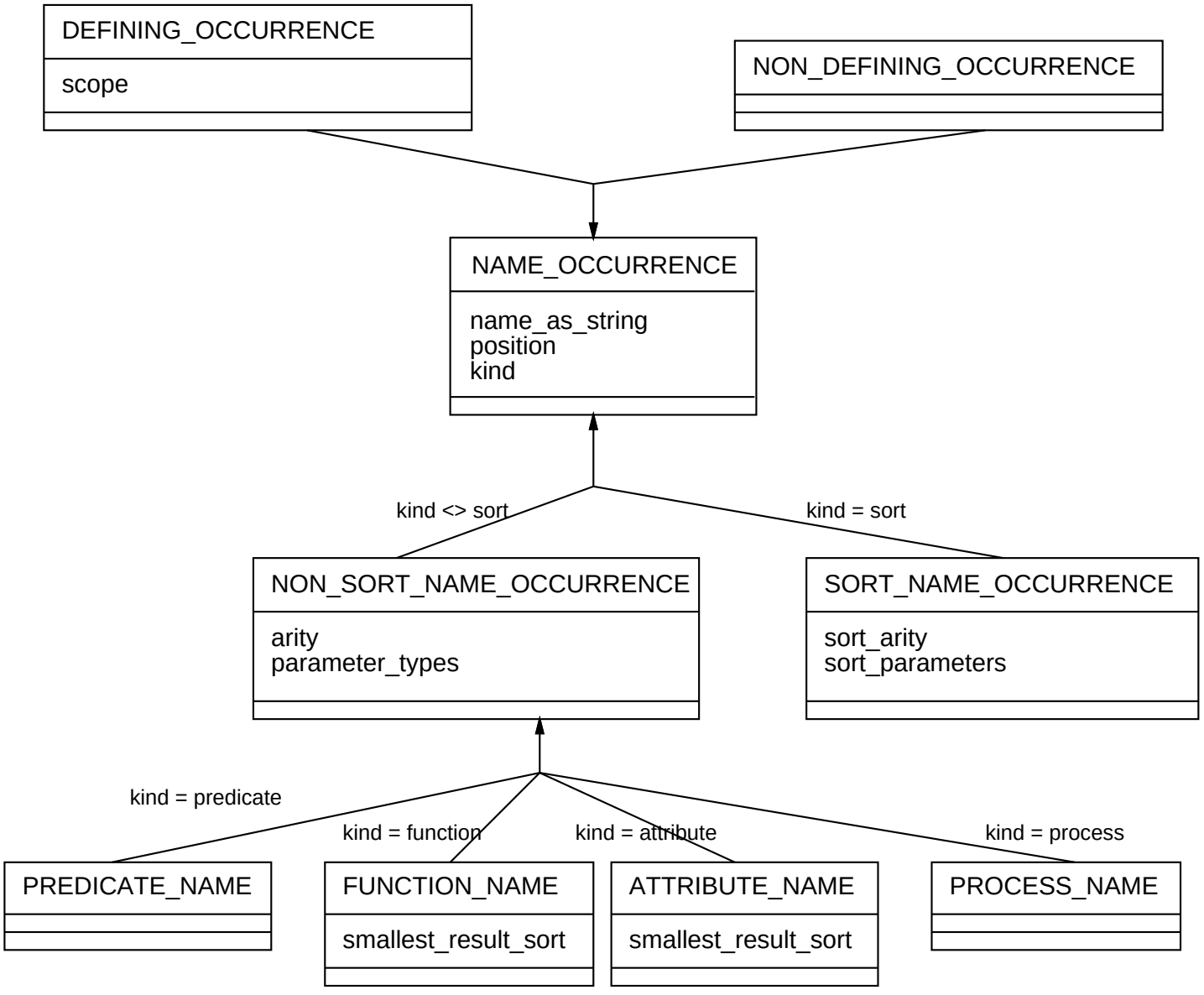

Figure 3.1: An overview of the concepts relating to names and their occurrences.

3.5 Determining the defining occurrence for an occurrence of a name

Now we are ready to state the rules that a defining occurrence o_d must satisfy in order to be called a defining occurrence for an occurrence o.

> **Unique definition constraint:** We require that for every occurrence of a name in a specification there is exactly one defining occurrence for that name.

If this constraint is violated, we are dealing with an ambiguous or incomplete specification. Finding more than one defining occurrence signals an ambiguity. If there is no defining occurrence, a name is used without being declared. [1]

The rules are as follows:

- If o itself is a defining occurrence, it is also a defining occurrence for o
- Otherwise, let o be a non-defining occurrence of name n. A defining occurrence o_d of a name is a defining occurrence for o iff all of the following conditions are met
  1. $Scope(o_d)$ contains $Position(o)$,
  2. o_d is an occurrence of name n,
  3. there is no alternative defining occurrence o_a that has a smaller scope than o_d, that satisfies rules 1 and 2 above (with o_d replaced by o_a).
  4. $Kind(o) = Kind(o_d)$,
  5. if $Kind(o_d) \in \{\mathsf{function}, \mathsf{attribute}, \mathsf{predicate}, \mathsf{process}\}$, then $Arity(o_d) = Arity(o)$ and $ParameterTypes(o) \leq ParameterTypes(o_d)$. The $\leq$ is the (non-strict) subsort relationship, extended to sequences of sorts (see section 5.5).
  6. if $Kind(o_d) \in \{\mathsf{function}, \mathsf{attribute}, \mathsf{predicate}, \mathsf{process}\}$, there is no alternative o_a to o_d with $ParameterTypes(o_a) < ParameterTypes(o_d)$, that satisfies rules 1,2,3,4 and 5 above. Note that $<$ denotes the *strict* subsort relationship extended to sequences of sorts (see section 5.5).
  7. if $Kind(o_d) = \mathsf{sort}$, then o_d should match o. This is described in more detail in section 5.4.

For example, as discussed in section 6.3, the occurrence q of the ⟨Name⟩ `Foo` in the ⟨Attributes⟩ section of an ⟨ObjectClassBlock⟩

```
begin object class WIDGET
  attributes
    Foo : BAR;
end object class WIDGET;
```

is a defining occurrence of a name. To be more specific, it is a defining occurrence of the name `Foo`, with $Scope(q)$ equal to the whole specification, $Arity(q) = 1$, $ParameterTypes(q) = \texttt{WIDGET}$, and $SmallestResultSort(q) = \texttt{BAR}$.

The $SmallestResultSort(q)$ property of occurrences of attribute and function names will only be described for defining occurrences q; non-defining occurrence have the same value for this property as their defining occurrence.

In the rest of this text we will only discuss LCM specifications that satisfy the unique definition constraint, and will talk about *the* defining occurrence for an occurrence of a name, since it has exactly one such defining occurrence.

[1] This constraint, combined with the global scope of many of the defining occurrences of name may turn out to be inconvenient when handling large LCM specifications. A solution might be introducing a syntactic modularization construction combined with a rule similar to the origin rule in ASF[2]. Modularization above the level of objects in CM specifications is subject of current research. For the moment, we will stick to this simple scheme.

Chapter 4

The global structure of LCM specifications

A LCM specification consists of definitions of ADTs, object classes, relationship classes and a descriptions of the atomic DBS transactions. Each of these kinds of definition has its own syntactic structure: the ⟨ValueBlock⟩, ⟨ObjectClassBlock⟩, ⟨RelationshipClassBlock⟩ and ⟨ServiceBlock⟩ respectively.

In order to increase readability of LCM specifications these blocks should be listed ordered such that all ADT specification are listed first, then all object and relationship class blocks, and all service blocks come last.

Syntax:

⟨LCM-Specification⟩ ⟶ ⟨ValueBlock⟩* { ⟨ObjRelBlock⟩ }* ⟨ServiceBlock⟩*

⟨ObjRelBlock⟩ ⟶ ⟨ObjectClassBlock⟩ | ⟨RelationshipClassBlock⟩

Occurrences of the same kind of block may be ordered as desired, but practice suggests that sorting in alphabetical order on the name of the entity is a useful convention.

In the next sections, we will look at each of these blocks in more detail.

Chapter 5

Values

In order to work conveniently with values, LCM has definitions of common value types (also commonly called datatypes) and some operators on them built-in. In addition, the user may supply his own definitions of Abstract Data Types (ADT's) in the framework of order sorted algebraic specifications, in a way similar to the specification languages OBJ[6] and ASF[2].

The syntactical construct for defining ADTs in LCM is the ⟨ValueBlock⟩, in which you bind a name to a set of values, and define the appearance of functions and constants related to that set. The meaning of the expressions using values from that set is expressed by giving equations.

As a simple example, a value block for specifying a sort representing the colors of a traffic light, together with a function `next_color` might look like this:

```
begin value type TRAFFIC_LIGHT_COLOR
  functions
    -- Constants are viewed as functions without parameters.
    red    : TRAFFIC_LIGHT_COLOR;
    orange : TRAFFIC_LIGHT_COLOR;
    green  : TRAFFIC_LIGHT_COLOR;

    -- A function to return the next color to be displayed,
    -- given the current color of a traffic light.
    next_color(TRAFFIC_LIGHT_COLOR) : TRAFFIC_LIGHT_COLOR;
  predicates
    continue(TRAFFIC_LIGHT_COLOR);
  axioms
    next_color(red) = green;
    next_color(orange) = red;
    next_color(green) = orange;
    -- Commonly encountered Amsterdam interpretation of traffic lights...
    forall c : TRAFFIC_LIGHT_COLOR ::
      continue(c) <-> c=green or c=orange;
end value type TRAFFIC_LIGHT_COLOR;
```

5.1 Names of built-in and user-defined value types

Both built-in and user-defined value types are refered to by mentioning their reserved word or name. The names of the built-in value types are predefined names of the language, and may not be redefined.
The internal identifier sorts that are generated for object classes in the absence of an explicitly defined identifier sort can also be refered to by means of a special sort expression discussed in subsection 5.4.

⟨SortName⟩ ⟶ ⟨PredefinedSortName⟩ | ⟨UserDefinedSortName⟩

⟨PredefinedSortName⟩ ⟶ `ZERO` | `POSINT` | `NEGINT` | `INATURAL` | `NZINTEGER` | `NATURAL` | `INTEGER` |
`NZRATIONAL` | `RATIONAL` | `DATE` | `CHARACTER` | `EMPTY` | `STRING` | `SET` |
`BAG` | `SEQUENCE` | `DATE`

⟨UserDefinedSortName⟩ ⟶ ⟨Name⟩

In the next sections we will first discuss user-defined value types. An overview of the built-in value types is given in section 5.10.

5.2 Value block

A **value block** groups together a number of value type definitions. Its syntax is as follows:

⟨ValueBlock⟩ ⟶ `begin value type` ⟨UserDefinedSortName⟩ *[* ⟨ValueBlockParameterList⟩ *]*
⟨ValueGeneralization⟩*
⟨ValueSpecialization⟩*
[⟨Functions⟩ *]*
[⟨ValuePredicates⟩ *]*
[⟨Axioms⟩ *]*
`end value type` ⟨UserDefinedSortName⟩ ;

⟨ValueBlockParameterList⟩ ⟶ `[` { ⟨ValueBlockParameter⟩ `,` }+ `]`

⟨ValueBlockParameter⟩ ⟶ `value type` ⟨UserDefinedSortName⟩ |
`function` ⟨UserDefinedFunctionName⟩ *[* ⟨FormalParameterList⟩ *]*
: ⟨SortExpression⟩ |
`predicate` ⟨UserDefinedPredicateName⟩ *[* ⟨FormalParameterList⟩ *]*

⟨UsedDefinedSortName⟩ ⟶ ⟨Name⟩

⟨UserDefinedFunctionName⟩ ⟶ ⟨Name⟩

⟨UserDefinedPredicateName⟩ ⟶ ⟨Name⟩

The ⟨UserDefinedSortName⟩s in the `begin value type` and `end value type` clauses must be the same. This is an instance of the

> **Block name rule**: the names occurring in the opening and closing phrases of a block must be identical.

We will encounter other instances of this rule in the discussion of the other block types of LCM.

An occurrence p of a name n as ⟨UserDefinedSortName⟩ immediately after the `begin value type` is a defining occurrence of name n, with $Kind(p)$ = sort and $Scope(p)$ being the complete LCM specification.

5.3 Parametrization of value blocks

$SortArity(p)$ is the number of occurrences of ⟨ValueBlockParameter⟩s that appear in the ⟨ValueBlockParameterList⟩, or zero if there is no ⟨ValueBlockParameterList⟩. $SortParameters(p)$ consists of the occurrences of ⟨UserDefinedSortName⟩, ⟨UserDefinedFunctionName⟩ and ⟨UserDefinedPredicateName⟩ in the order they occur as ⟨ValueBlockParameter⟩ in the ⟨ValueBlockParameterList⟩. All these occurrences o are defining occurrences with $Type(o)$ equal to sort, function and predicate, respectively. $Scope(o)$ is the whole ⟨ValueBlock⟩ in which the defining occurrence is found. If $Type(o) =$ sort, then $SortArity(s) = 0$.

Example

The definition of a parametrized value type for arrays parametrized by value types for the indices and the elements, together with a predicates that determines whether two indices are equal may be defined as

```
begin value type ARRAY [value type INDEX,
                        value type ELEMENT,
                        predicate IndexEqual(INDEX,INDEX)]
  functions
    ...
  predicates
    ...
  axioms
    ...
end value type ARRAY;
```

The scope of the defining occurrences of the names `INDEX`, `ELEMENT` and `IndexEqual`, found between the square brackets is the whole ⟨ValueBlock⟩ for `ARRAY`. The defining occurrence of name `ARRAY` has the complete LCM source text as its scope. We might use this name somewhere else in the specification to refer to the data type, e.g.

```
  ...
  attributes
    owners : ARRAY[INTEGER, OWNER, same_owner(OWNER,OWNER)];
  ..
```

appearing in an ⟨ObjectClassBlock⟩.

As a consequence of the unique definition constraint, there must appear exactly one ⟨ValueBlock⟩ for a given ⟨UserDefinedSortName⟩ in a LCMspecification. Thus, we may not have a LCM specification like

```
begin value type Foo
  ...
end value type Foo;
...
begin value type Foo
  ...
end value type Foo;
```

There is nothing wrong with a name of a ⟨ValueBlock⟩ reappearing as the name of an ⟨ObjectClassBlock⟩, as in

```
begin value type Foo
  ...
end value type Foo;
  ...
```

```
begin object class Foo
  ...
end object class Foo;
```

This has the special meaning that values of value type `Foo` are used as internal object identifiers for the objects of class `Foo` (see also section 6.1).

This special case only applies to object classes and not to relationship classes; thus, there must not be a ⟨RelationshipClassBlock⟩ with the same name as a ⟨ValueBlock⟩.

The intended semantics of parametrized value blocks is based on a normalization process on the LCM source text that replaces the parametrized value type block by many instances, one for each sort expression of that parametrized sort with a particular set of parameters, with the formal parameters of the instances replaced by the actual parameters in the sort expression. This will not be described in more detail here; see the description of a similar mechanism in ASF[2].

5.4 Sort expressions

In order to refer to a sort ⟨SortExpression⟩s are used. There are two forms of ⟨SortExpression⟩s. The first form consists of mentioning a sort name, possibly with parameters used to instantiate a parametrized ⟨ValueBlock⟩. The second form consists of a sequence of class names concatenated with an asterisk, as in `A*B*C`. The intended meaning of this sort expression is the sort that is the intersection of the sorts `A`,`B` and `C`. This form may only be used to refer to implicitly defined internal identifier sorts that are an intersection of object subclasses.

This facility is needed because creation and deletion events can only be defined for species. A **species** is a class such that it is the smallest class of its instances. In the presence of multiple partitionings of an object class there may not be an explicit name for the intersection of subclasses from different partitionings. For example, in the following specification

```
begin object class A
  partitioned by B, C;
  partitioned by D, E;
end object class A;
```

the species of all A's that are B's but are also D's (but not C's and not E's) doesn't have a name, and similarly so for the three other combinations. This class structure and the intersections are shown in figure 5.1. To create objects of A, we need to define creation events for these species, but we can only reference these species by describing them as the intersection of subclasses from different partitionings of `A`, e.g. as `B*D` for the `A`'s that are both `B`'s and `D`'s:

```
  ...
  events
    create(B*D; FOO, BAR);
  axioms
    forall elt : B*D, f : FOO, b : BAR ::
      [create(elt;f,b)] ...
  ...
```

Syntax:

⟨SimpleSortExpression⟩ ⟶ ⟨SortName⟩ *[* ⟨ActualValueBlockParameterList⟩ *]*

⟨SortExpression⟩ ⟶ { ⟨SimpleSortExpression⟩ * }+

⟨ActualValueBlockParameterList⟩ ⟶ [{ ⟨ActualValueBlockParameter⟩ , } +]

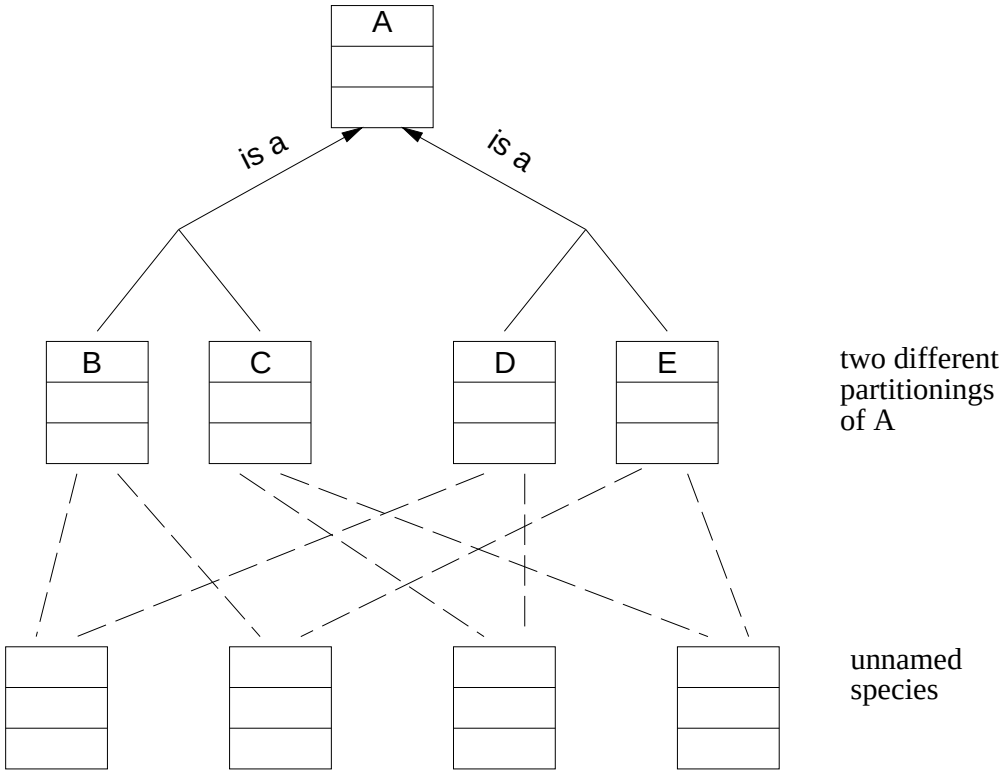

Figure 5.1: Class A is partitioned into B and C according to one partition and into D and E according to another. To create an object of class A that is both a B and a D, we need to refer to its species, which is the unnamed intersection of classes B and D.

⟨ActualValueBlockParameter⟩ ⟶ ⟨SortExpression⟩ |
`function` ⟨PrefixForm⟩ : ⟨SortExpression⟩ |
`predicate` ⟨PrefixForm⟩

An occurrence q of a name as ⟨SortName⟩ in a ⟨SimpleSortExpression⟩ has $Kind(q)$ = sort, $SortArity(q)$ = the number of ⟨ActualValueBlockParameter⟩s in its ⟨ActualValueBlockParameterList⟩, or zero if the ⟨SortExpression⟩ contains no ⟨ActualValueBlockParameterList⟩. If $SortArity \neq 0$, $SortParameters(q)$ consists of the occurrences of the names as ⟨SortName⟩ of a ⟨SortExpression⟩ or ⟨UserDefinedPrefixName⟩ of an ⟨ActualValueBlockParameter⟩ in the order in which they appear in the source text.

We will say that an occurrence of a name as ⟨SortName⟩ in a sort expression q **matches** a defining occurrence d of a name that is a sort name if

1. $SortArity(q) = SortArity(d)$, and
2. for each formal parameter occurrence f_d in $SortParameters(d)$ and its corresponding actual value block parameter f_q:
   - If $Kind(f_d)$ = sort then the sort expression f_q given as actual parameter should be a (non-strict) subsort of the sort given as actual parameter.
   - If $Kind(f_d)$ = function or $Kind(f_d)$ = predicate then $ParameterTypes(f_q) \leq ParameterTypes(f_d)$.

   the sorts and parameter types of the actual value block parameters and substituting the respective $SortParameters(q)$ for $SortParameters(d)$ will not cause a sort conflict.

This is part of the requirement for a sort name to be a defining occurrence for another name, as described in section 3.5.

5.5 Generalization and specialization of value types

A **value specialization section** expresses that the value type of the ⟨ValueBlock⟩ in which it appears has the types mentioned in the ⟨ValueSpecialization⟩ as (not necessarily proper) subtypes. Its syntax is:

⟨ValueSpecialization⟩ ⟶ `specialized by` { ⟨SubSortExpression⟩ `,` }+ `;`

⟨SubSortExpression⟩ ⟶ ⟨SortExpression⟩

The subtype structure induced by all ⟨ValueSpecialization⟩s of the specification taken together should satisfy the

> **Non-circularity constraint**: There doesn't exist a (possibly empty) sequence of ⟨SubSortExpression⟩s $S_1, \ldots, S_n$ such that `specialized by` S_2`,`... appears in the ⟨ValueBlock⟩ for S_1, and `specialized by` S_3`,`... appears in the ⟨Valueblock⟩ for S_2, and so on to `specialized by` S_n`,`... appearing in the ⟨ValueBlock⟩ of S_{n-1}. Note that this also excludes the immediate case of a ⟨ValueBlock⟩ for `S` containing `specialized by S;`.

The value specialization sections together define a relation on sort expressions. The transitive reflexive closure of this relation is called the **subsort** or **subtype ordering** on sort expressions, written as $\leq$. Sometimes we use $<$ to denote the transitive closure of the relation on sort expressions defined by the value specialization sections. Both relations are extended to sequences of sort expression such that they hold if it holds for all elements of the sequences, which should be of the same length.

A **value generalization section** expresses that the sort specified in this *ValueBlock* is a (not necessarily proper) subsort of other sorts. Its syntax is:

⟨ValueGeneralization⟩ ⟶ `specialization of` { ⟨SuperSortName⟩ `,` }+ `;`

⟨SuperSortName⟩ ⟶ ⟨UserDefinedSortName⟩

Note that you cannot declare a user-defined value type to be a subsort of a built-in datatype[1].

A ⟨ValueGeneralization⟩, if added (it is optional), should be consistent with the ⟨ValueSpecialization⟩s in the sense that if a ⟨ValueGeneralization⟩ is present in a value block, it is required that a corresponding ⟨ValueSpecialization⟩ appears in the ⟨ValueBlock⟩ of the supertype of that block. In a sense, a ⟨ValueGeneralization⟩ is a kind of comment which is checked for consistency with the rest of the specification. Given the availability of the ⟨ValueSpecialization⟩, the ⟨ValueGeneralization⟩ is superfluous. This redundancy was introduced on purpose to aid the human reader in understanding the individual ⟨ValueBlock⟩s by providing more context about its location in the type hierarchy. A ⟨ValueGeneralization⟩ reminds the human reader that this value type is a subtype of some set of other types.

The syntax of ⟨ValueSpecialization⟩ is a special case of the syntax found for specialization in the case of objects and relationships, where additional information can be provided to express properties of disjointness of the subtypes and exhaustiveness of the enumeration of subtypes (section 6.2).

Example

```
begin value type DAY
  specialized by WORKING_DAY, WEEKEND_DAY;
end value type DAY;

begin value type WORKING_DAY
  specializing DAY;
  functions
    Monday    : WORKING_DAY;
    Tuesday   : WORKING_DAY;
    Wednesday : WORKING_DAY;
    Thursday  : WORKING_DAY;
    Friday    : WORKING_DAY;
end value type WORKING_DAY;

begin value type WEEKEND_DAY
  -- mentioning the supertype, as is done in WORKING_DAY,
  -- is optional, and omitted here.
  functions
    Saturday  : WEEKEND_DAY;
    Sunday    : WEEKEND_DAY;
end value type WEEKEND_DAY;
```

5.6 Functions section

The signature of the order-sorted algebra used to model the ADT is given in the **functions section**.

It allows the definition of prefix functions, with arguments in parentheses.

For convenience, infix and mixfix operators with the expected associativity and priority are available for use with the commonly used built-in value types. Their names and syntax are a part

[1]Later versions of LCM are likely to be more permissive in this respect, but to do this well we'll probably need sort constraints as well (currently not included in LCM) to avoid problems due to non-conservative extension of the built-in datatypes by user-defined subsorts of it.

of the language definition (see section 5.10). Currently, these features provide more freedom of syntax than is available for user defined value types.

Constants are considered to be functions without arguments, and thus should be declared in the functions section too.

The syntax is:

⟨Functions⟩ —→ `functions` ⟨FunctionDeclaration⟩*

⟨FunctionDeclaration⟩ —→ ⟨PrefixForm⟩ : ⟨SortExpression⟩ ;

⟨PrefixForm⟩ —→ ⟨Name⟩ *[* ⟨FormalParameterList⟩ *]*

⟨FormalParameterList⟩ —→ ({ ⟨FormalParameter⟩ , }+)

⟨FormalParameter⟩ —→ ⟨SortExpression⟩

The signature obtained by combining all ⟨FunctionDeclation⟩s, ⟨ValueSpecialization⟩s, taking into account the (implicit) definition of internal identifier sorts and their subsort structure (which follows the specialization of objects and relationships) and ⟨AttributeSpecification⟩s of a LCM specification should be **regular**, which is defined as follows (based on a definition by Smolka et al. [10]):

A signature Σ is called regular if

1. the subsort order of Σ is a partial order
2. every ⟨Expression⟩ s has a least sort σs, that is, there is a unique function σ from the set of all well-formed ⟨Expression⟩s into the set of ⟨SortName⟩s such that
   - if s is an ⟨Expression⟩, then s is a ⟨Expression⟩ of sort σs
   - if s is an ⟨Expression⟩ of sort ξ, then $\sigma s \leq \xi$.

   We call a specification regular if its signature is regular.

Example

A ⟨Functions⟩ section might look like

```
functions
  first_working_day : WORKING_DAY;
  next_working_day(WEEK_DAY) : WORKING_DAY;
  remaining_working_days(WEEK_DAY) : SET[WORKING_DAY];
```

The following is not a valid ⟨Functions⟩ section, because it violates the regularity requirement (assuming that the sorts `MyList` and `MyTree` are unrelated in the sort hierarchy):

```
begin value type MyList
  functions
    null : MyList; -- null returns the empty list
    ...
end value type MyList;

begin value type MyTree
  functions
    null : MyTree; -- null returns the empty tree
    ...
end value type MyTree;
```

Here, the ⟨Expression⟩ `null` has `MyList` and `MyTree` as sorts. Since we assumed that neither of these is smaller than the other, the expression `null` has no unique least sort, thus violating the regularity requirement.

5.7 Predicates section

It is possible to define predicates on user-defined value types, using the following syntax:

⟨ValuePredicates⟩ ⟶ `predicates` ⟨ValuePredicateDeclaration⟩*

⟨ValuePredicateDeclaration⟩ ⟶ ⟨PrefixForm⟩ ;

Although one could in principle define constant predicates with this syntax, the built-in predicate expressions `True` and `False` are more convenient (see section 5.10).

Note that predicates on value types (⟨ValuePredicate⟩s) allow an arbitrary number of parameters, whereas predicates on object and relationship classes (see 6.9) are always unary predicates (one argument is assumed, which is an internal identifier for an element of the object or relationship class in which they are defined).

Example

```
predicates
  adjacent_working_days(WORKING_DAY, WORKING_DAY);
```

5.8 Axioms section

Syntax:

⟨Axioms⟩ ⟶ `axioms` ⟨Axiom⟩*

⟨Axiom⟩ ⟶ *[* ⟨Quantification⟩ `::` *]* ⟨AxiomBody⟩ ;

⟨Quantification⟩ ⟶ `forall` { ⟨VariableTyping⟩ `,` }+

⟨VariableTyping⟩ ⟶ { ⟨VariableName⟩ `,` }+ `:` ⟨SortExpression⟩

⟨AxiomBody⟩ ⟶ *[* ⟨Condition⟩ `->` *]* ⟨Condition⟩

⟨AxiomBody⟩ ⟶ *[* ⟨Condition⟩ `<->` *]* ⟨Condition⟩

⟨Condition⟩ ⟶ ⟨SimpleCondition⟩

⟨SimpleCondition⟩ ⟶ { ⟨PredicateTerm⟩ `,` }+

⟨PredicateTerm⟩ ⟶ { ⟨PredicateFactor⟩ `or` }+

⟨PredicateFactor⟩ ⟶ { ⟨PredicateLiteral⟩ `and` }+

⟨PredicateLiteral⟩ ⟶ *[* `not` *]* ⟨PredicateApplication⟩

⟨PredicateApplication⟩ ⟶ ⟨PrefixPredicateApplication⟩ | ⟨InfixPredicateApplication⟩

⟨PrefixPredicateApplication⟩ ⟶ ⟨PrefixPredicateName⟩ *[* ⟨ActualParameterList⟩ *]*

⟨PrefixApplication⟩ is applicable to user-defined predicates as well as the predefined prefix predicates such as `Exists` and `True`. ⟨InfixPredicateApplication⟩s are only possible for the predefined predicates, see subsection 5.10

The comma between ⟨PredicateTerm⟩s means a logical "and".

This syntax is very permissive in ⟨AxiomBody⟩: it allows a mixture of equations, unequalities combined with a logic programming notation for predicates and conditions. It is likely that an implementation will restrict this further. The exact nature of these restrictions is left undefined here. The reason for this permissiveness is that we reuse this grammar rule for the axioms section in the definition of classes. We don't want to be unnecessarily restrictive in those specifications.

Example

Part of an axioms section defining a value type `BOOLEAN`, with values `true` and `false` and the functions `conjunction` and `negation` etc. might look like [2]

```
axioms
  forall x : BOOLEAN ::
    conjunction(true, x) = true;
  forall x : BOOLEAN ::
    conjunction(false, x) = false;
  negation(true) = false;
  negation(false) = true;
  forall x, y : BOOLEAN ::
    disjunction(x,y) = negation(conjunction(negation(x),negation(y)));
  ...
```

5.9 Expressions

An expression is constructed from function applications of user defined and built-in functions. We call an expression well-formed if the arguments supplied to each application fit with exactly one definition of that function. This is guaranteed by the unique definition rule and the syntax rules given below. The predefined functions are considered to have an implicit defining occurence in the specification, with the arguments as indicated below. Of course, the unique definition rule applies to uses of the built-in functions as well as to user-defined functions.

⟨Expression⟩ ⟶ ⟨SimpleExpression⟩ { ⟨BinaryInfixOperator⟩ ⟨SimpleExpression⟩ }*

⟨SimpleExpression⟩ ⟶ ⟨PrefixFunctionApplication⟩

[2] Note that `BOOLEAN` is not a built-in value type; it is often more convenient to use predicates instead. Since `True` (note the capital 'T'!) , `False` and operators `and` etc. are reserved words LCM dealing with predicates, the names used in the example are somewhat unnatural.

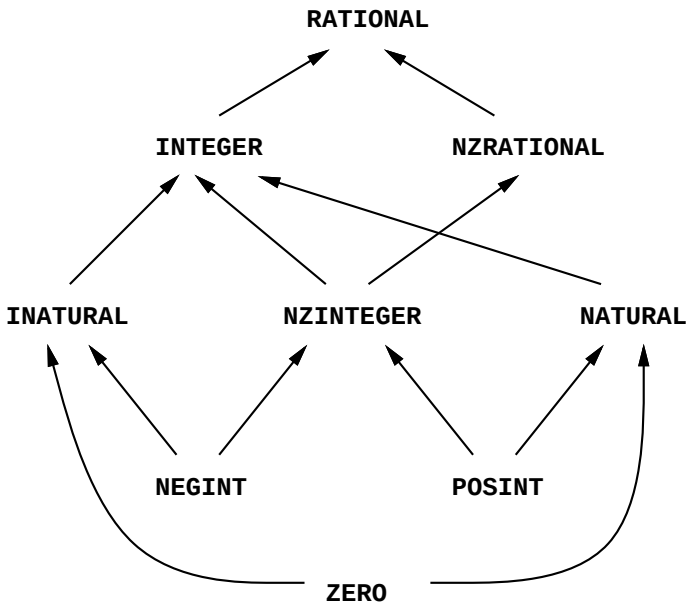

Figure 5.2: The built-in value types, and the subsort relationships between them.

⟨SimpleExpression⟩ ⟶ (⟨Expression⟩)

⟨SimpleExpression⟩ ⟶ ⟨Constant⟩

⟨SimpleExpression⟩ ⟶ ⟨VariableName⟩

⟨SimpleExpression⟩ ⟶ - ⟨SimpleExpression⟩

⟨PrefixFunctionApplication⟩ ⟶ ⟨PrefixFunctionName⟩ *[* ⟨ActualParameterList⟩ *]*

⟨ActualParameterList⟩ ⟶ ({ ⟨Expression⟩ , }+)

⟨VariableName⟩ ⟶ ⟨Name⟩

For a survey of the ⟨BinaryInfixOperator⟩s and the ⟨PrefixFunctionName⟩s for the predefined functions, see section 5.10.

5.10 Built-in value types, predicates and functions

To aid the author in writing LCM specifications, a small set of built-in value types is provided, with operators and predicates defined on them. The built-in value types and the subsort relationships between them are shown in figure 5.2.

5.10.1 Built-in predicates for testing equality

An important class of predicates is that of equality. Two variants are provided: the first states that two expressions are equal, the other that they are unequal.

Predicate	Description
`True`	Constant true
`False`	Constant false

Figure 5.3: The built-in nullary predicates.

Predicate	Description
`Exists(instance_id)`	Instance existence. See section 8.9
`Used(instance_id)`	Instance identifier used. See section 8.9
`IsEmpty(SET[t])`	Is an empty set of elements of type t
`IsEmpty(BAG[t])`	Is an empty bag of elements of type t

Figure 5.4: The built-in unary predicates that use standard predicate application syntax.

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ = ⟨Expression⟩

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ != ⟨Expression⟩

The types of the expressions appearing on both sides of the = or != must have a unique least common supertype.

The desired semantics of specifications containing both equality and disequality predicates needs further study, in particular the exact constraints that should be imposed in order to make executability feasible. In this report we deal only with syntactic matters and we leave this matter open.

5.10.2 Other built-in predicates

In figure 5.3, the two predefined predicates without arguments, `True` and `False` are shown.

Both nullary and unary predefined predicate names use the standard syntax for a ⟨PrefixPredicateApplication⟩.

Figure 5.5 shows the binary built-in predicates with infix notation.

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ ⟨InfixPredicate⟩ ⟨Expression⟩

Here, the ⟨InfixPredicate⟩, can be any of the reserved words mentioned in the first column of figure 5.5.

5.10.3 Built-in sorts, constants and functions

The built-in expression forms consist of constants, the unary negation operator and a number of infix binary operators.

⟨Constant⟩ ⟶ ⟨ZeroConstant⟩ | ⟨PosIntConstant⟩ | ⟨RationalConstant⟩ | ⟨CharacterConstant⟩ | ⟨StringConstant⟩

All occurrences c of ⟨ZeroConstant⟩, ⟨PosIntConstant⟩, ⟨RationalConstant⟩, ⟨CharacterConstant⟩ and ⟨StringConstant⟩ are occurrences of built-in functions with $Arity(c) = 0$, and $SmallestResultSort(c)$ equal to `ZERO`, `POSINT`, `RATIONAL`, `CHARACTER`, and `STRING`, respectively.

Expressions may be combined into larger expressions with the ⟨BuiltInInfixOperator⟩s shown in figure 5.6. All these binary functions are left-associative. Their priorities are summarized in figure 5.7.

Expressions may be combined into larger expressions using the built-in functions shown in figure 5.8; the are described by giving the ⟨FunctionDeclaration⟩s that would be needed to declare them.

They are written just like ordinary user-defined function applications. The author of a LCM specification may assume them to be defined somewhere.

Predicate	LHS type	RHS type	Description
<	NATURAL	NATURAL	strictly smaller than
<	INT	INT	strictly smaller than
<	RATIONAL	RATIONAL	strictly smaller than
<	SET[t]	SET[t]	strict set inclusion
<	STRING	STRING	strictly precedes
<	DATE	DATE	strictly precedes
<=	NATURAL	NATURAL	smaller than or equal
<=	INT	INT	smaller than or equal
<=	RATIONAL	RATIONAL	smaller than or equal
<=	SET[t]	SET[t]	non-strict set inclusion
<=	STRING	STRING	non-strictly precedes
<=	DATE	DATE	non-strictly precedes
>=	NATURAL	NATURAL	greater than or equal
>=	INT	INT	greater than or equal
>=	RATIONAL	RATIONAL	greater than or equal
>=	SET[t]	SET[t]	is non-strict superset of
>=	STRING	STRING	non-strictly succeeds
>=	DATE	DATE	non-strictly succeeds
>	NATURAL	NATURAL	strictly greater than
>	INT	INT	strictly greater than
>	RATIONAL	RATIONAL	strictly greater than
>	SET[t]	SET[t]	is strict superset of
>	STRING	STRING	strictly succeeds
>	DATE	DATE	strictly succeeds
In	t	SET[t]	Membership of a set

Figure 5.5: Built-in binary predicates that use infix syntax.

Operator	LHS type	RHS type	Result type	Description
+	ZERO	ZERO	ZERO	Addition
+	NATURAL	NATURAL	NATURAL	Addition
+	INTEGER	INTEGER	INTEGER	Addition
+	RATIONAL	RATIONAL	RATIONAL	Addition
+	SET[t]	SET[t]	SET[t]	Set union
+	BAG[t]	BAG[t]	BAG[t]	Bag union
-	ZERO	ZERO	ZERO	Subtraction
-	NATURAL	NATURAL	NATURAL	Subtraction
-	INTEGER	INTEGER	INTEGER	Subtraction
-	RATIONAL	RATIONAL	RATIONAL	Subtraction
-	SET[t]	SET[t]	SET[t]	Set difference
-	BAG[t]	BAG[t]	BAG[t]	Bag difference
-	DATE	DATE	INTEGER	Difference (in number of days)
div	NAT	POSINT	NAT	Integer division
mod	NAT	POSINT	NAT	Integer remainder
^	NAT	NAT	NAT	Exponentiation
*	ZERO	ZERO	ZERO	Multiplication
*	NATURAL	NATURAL	NATURAL	Multiplication
*	INTEGER	INTEGER	INTEGER	Multiplication
*	RATIONAL	RATIONAL	RATIONAL	Multiplication
*	SET[t]	SET[t]	SET[t]	Set intersection
*	BAG[t]	BAG[t]	BAG[t]	Bag intersection
*	SEQUENCE[t]	SEQUENCE[t]	SEQUENCE[t]	Sequence concatenation
*	STRING	STRING	STRING	String concatenation
/	RATIONAL	NZRATIONAL	NZRATIONAL	Division

Figure 5.6: Built-in LCM binary functions that use infix syntax. All binary functions mentioned above are left associative.

- (unary operator)
^
*, /, div
+, - (binary operator)

Figure 5.7: Priority of built-in unary and binary functions. Reading from top to bottom, groups of operators with equal priority are listed in order of decreasing priority.

Declaration	Description
`inc(NATURAL):POSINT;`	increment, successor
`inc(NEGINT):INAT;`	
`inc(INTEGER):INTEGER;`	
`dec(POSINT):NATURAL;`	decrement, predecessor
`dec(INAT):NEGINT;`	
`dec(INTEGER):INTEGER;`	
`gcd(NZINTEGER,NZINTEGER):NZINTEGER;`	greatest common divisor
`integer_part(RATIONAL):INTEGER;`	integer part of a decimally represented rational
`decimal_part(RATIONAL):NATURAL;`	decimal part of a decimally represented rational
`floor(RATIONAL):INTEGER;`	largest integer $\leq$ the rational
`empty:EMPTY;`	empty container (set, bag, list etc.) constant
`insert(SET[t],t):SET[t];`	include element in a set
`insert(BAG[t],t):BAG[t];`	add one occurrence of element to a bag
`delete(SET[t],t):SET[t];`	exclude element from a set
`delete(BAG[t],t):BAG[t];`	remove one occurrence of element from a bag
`size(SET[t]):NATURAL`	cardinality of a set
`size(BAG[t]):NATURAL`	number of occurrences in a bag
`size(SEQUENCE[t]):NATURAL`	length of a sequence
`size(STRING):NATURAL`	number of characters in a `STRING`
`bag_to_set(BAG[t]) : SET[t];`	set of all elements occurring in the bag
`set_to_bag(SET[t]) : BAG[t];`	bag, with all elements of set occurring once
`string_to_date(STRING) : DATE;`	convert string yyyymmdd to a `DATE` value
`year(DATE) : NATURAL;`	extract year from a `DATE` value
`month(DATE) : NATURAL;`	extract month from a `DATE` value
`day(DATE) : NATURAL;`	extract day from a `DATE` value
`date_to_string(DATE) : STRING;`	convert a `DATE` to a string yyyymmdd

Figure 5.8: Built-in functions in LCM that use the standard function-call syntax.

Chapter 6

Objects

In chapter 5 we introduced value blocks, which fix particular sets of values and functions at specification time. We can do so because they will remain the same over all possible states of the UoD. On the other hand, we need to model change as well. In the next subsections we will explain the constructions for describing modeled entities that do not necessarily remain the same over all states of the world traversed during the life of the DBS: objects, attributes and relationships. Although we cannot fix the sets of all object instances or the relations between them at specification time, we can express restrictions on them and express similarities between them in different states of the UoD.

6.1 Object class definition

An **object class block** specifies a class by specifying the common structure of all possible class instances (the objects).

⟨ObjectClassBlock⟩ ⟶ `begin object class` ⟨UserDefinedObjectClassName⟩
⟨ObjectGeneralization⟩*
⟨ObjectSpecialization⟩*
[⟨Attributes⟩ *]*
[⟨Predicates⟩ *]*
[⟨Events⟩ *]*
[⟨LifeCycle⟩ *]*
[⟨Axioms⟩ *]*
`end object class` ⟨UserDefinedObjectClassName⟩ ;

⟨UserDefinedObjectClassName⟩ ⟶ ⟨Name⟩

The ⟨UserDefinedObjectClassName⟩s in the `begin object class` and `end object class` clauses must be identical. This is a second instance of the block name rule mentioned earlier in section 5.2.

There may be at most one ⟨ObjectClassBlock⟩ for a ⟨ObjectClassName⟩. There cannot be any ⟨RelationshipClassBlock⟩ in the specification with the same name as an ⟨ObjectClassBlock⟩.

There may however be a ⟨ValueClassBlock⟩ with a ⟨UserDefinedSortName⟩ identical to the ⟨UserDefinedObjectClassName⟩ of some ⟨ObjectClassBlock⟩. In this case, the specifier explicitly supplies the set of internal identifiers for the objects in the class. If this is done for a class, it is required that this is done for all its descendant classes as well. In the case no such value types are defined explicitly for an object class, internal identifier sorts for them are defined implicitly. This is described in detail in section 6.11.

Note that object class specifications cannot be parametrized. One reason for not allowing this is that we have not yet come across applications from which we could abstract an object class specification parametrized by attributes, events, or life cycles.

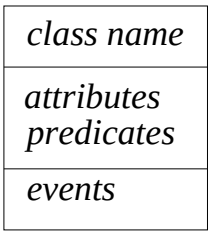

Figure 6.1: The general form of a class diagram, the diagrammatic summary of an object class block.

In the next section we discuss the **state signature** of an object class specification, i.e. the generalization, specialization, attributes and predicates sections. Events are discussed in chapter 8, and life cycles in chapter 9.

The diagrammatic representation of a classes shown in figure 6.1. This is called a **class diagram**.

6.2 Generalization and specialization of object classes

An **object specialization section** expresses that the object class of the ⟨ObjectClassBlock⟩ in which it appears has the object classes mentioned in the ⟨ObjectSpecialization⟩ as (not necessarily proper) subclasses. The subclasses are required to be disjoint, and, when taken together, should cover all elements of their superclass.

Its syntax is:

⟨ObjectSpecialization⟩ ⟶ `partitioned by` { ⟨ObjectSubClassName⟩ `,` }+ ;

⟨ObjectSubClassName⟩ ⟶ ⟨UserDefinedObjectClassName⟩

⟨ObjectGeneralization⟩ ⟶ `specialization of` { ⟨SuperClassName⟩ `,` }+ ;

⟨SuperClassName⟩ ⟶ ⟨UserDefinedClassName⟩

An extended subclass relationship is derived from all ⟨ObjectSpecialization⟩s of the specification taken together by taking the closure over transitivity of the subclass. This extended subclass relationship should be irreflexive. This is to be understood to also exclude statements of c having itself as subclass, even though this would seem reasonable under the non-proper subset of objects interpretation of subclassing. Together with this restriction, the syntax implies that there may be zero or more ⟨ObjectSpecialization⟩ sections in an ⟨ObjectClassBlock⟩, and each ⟨ObjectSpecialization⟩ section declares a partition of two or more disjoint subclasses. This means that we can have multiple inheritance. On the other hand, we cannot have a single subclass c_1 of c_0, but must always at least add another subclass c_2 of c_0 so that c_1 and c_2 form a partitioning of c_0.

Example

Consider the situation shown in in figure 6.2. In LCM, this could be described as sketched below:

```
begin object class PERSON
  partitioned by
    MALE, FEMALE;
```

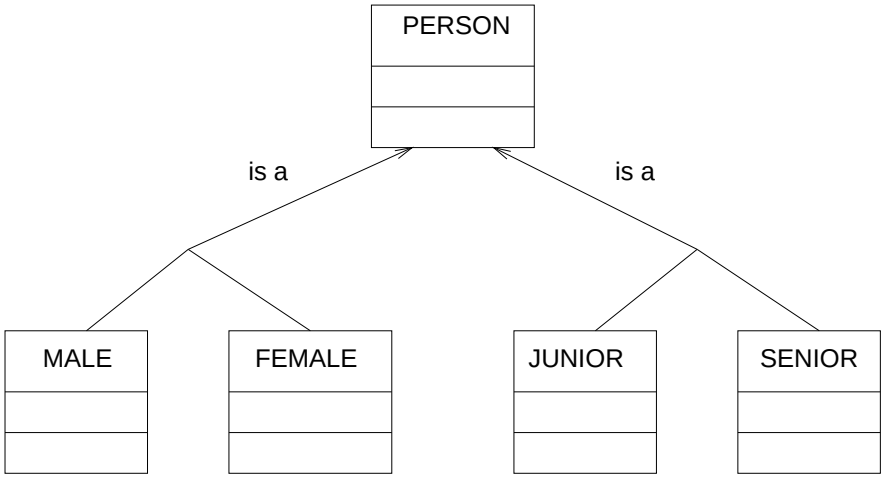

Figure 6.2: An example of an object specialization diagram.

```
  partitioned by
    JUNIOR, SENIOR;
end object class PERSON;

begin object class MALE
  ...
end object class MALE;

begin object class FEMALE
...
end object class FEMALE;

begin object class JUNIOR
...
end object class JUNIOR;

begin object class SENIOR
...
end object class SENIOR;
```

6.3 Attribute specification

In each state of the world, an attribute name a is interpreted as a function $a : D \rightarrow C$, where D is its domain and C is its codomain.. Therefore, within a state of the world, it behaves just like a `function`, but in different world states, one attribute name may be interpreted as different functions. The interpretation of an attribute may change when an event occurs. This is described in chapter 8.

Attributes are always declared inside a ⟨ObjectClassBlock⟩ or ⟨RelationshipClassBlock⟩ ant the domain of an attribnute is always the class in whose specifrication it is declared. In the following syntax, only the codomain name needs to be specified.

⟨Attributes⟩ ⟶ `attributes` { ⟨AttributeSpecification⟩ }* { ⟨UniquenessConstraint⟩ }*

⟨AttributeSpecification⟩ ⟶ ⟨AttributeName⟩ : ⟨SortExpression⟩ ⟨AttributeFlags⟩ ;

⟨AttributeFlags⟩ ⟶ { ⟨AttributeFlag⟩ , }*

⟨AttributeFlag⟩ ⟶ ⟨Initialization⟩

⟨AttributeFlag⟩ ⟶ `fixed`

⟨AttributeFlag⟩ ⟶ `inverse`

⟨AttributeFlag⟩ ⟶ `injection` | `surjection` | `bijection`

⟨UniquenessConstraint⟩ ⟶ `keys` ⟨KeyConstraint⟩*

⟨KeyConstraint⟩ ⟶ { ⟨AttributeName⟩ , }+ ;

⟨UniquenessConstraint⟩ ⟶ `identifiers` ⟨IdentifierConstraint⟩*

⟨IdentifierConstraint⟩ ⟶ ⟨AttributeName⟩ ;

See section 8.9.3 for a discussion of ⟨Initialization⟩.

Single attribute initialization rule: Only one initialization as attribute flag is allowed in an ⟨AttributeSpecification⟩, even if the values match.

A redefinition of the initial value is allowed in another ⟨ValueBlock⟩ for a subclass.

Flags, keys and identifiers are discussed in the following sections. We use the following example in what follows:

```
begin object class COPY
  attributes
    bar_code : INTEGER;
    price : MONEY fixed;
    times_borrowed : NATURAL initially 0;
  identifiers
    bar_code;
end object class COPY;
```

6.4 Fixed attributes

The flag `fixed` for an attribute `a` is syntactic sugar for the inclusion of axioms of the form
`forall x : S :: Exists(x), a(x) = v -> [e(x;...)]a(x) = v`
for all events `e` applicable to objects of class `S`.

The meaning of this is that if `x` exists and `a(x)=v` before the execution of `e`, then after execution of event `e(x;...)`, we have `a(x)=v`. These axioms say that, once an object exists, none of its fixed attributes changes under any event `e`. These axioms follow from the **frame assumption**, which says that everything that is not specified to change in an event remains unchanged.

6.5 Injective, surjective and bijective attributes

In a particular state of the world, an attribute can be viewed as a function. By stating a property like surjectivity of an attribute this function is constrained, thus giving an easy way to specify some common cardinality constraints, without having to add special axioms.

Figure 6.3: Each title is the title of at most one document.

Figure 6.4: There is at least one document for each title.

Injectivity

For example, consider the constraint that in each state of the world, there is at most one document for each title, as shown in figure 6.3. This can be expressed by:

```
begin object class DOCUMENT
  attributes
    title : TITLE injection;
  ...
end object class DOCUMENT;

begin object class TITLE
  ...
end object class TITLE;
```

This is equivalent to the inclusion of an explicit axiom stating

```
  forall d1, d2 : DOCUMENT ::
    title(d1) = title(d2) -> d1 = d2;
```

Surjectivity

There is at least one document for each title, as shown in figure 6.4:

```
begin object class DOCUMENT
  attributes
    title : TITLE surjection;
  ...
end object class DOCUMENT;

begin object class TITLE
  ...
end object class TITLE;
```

This is equivalent to the definition of an additional right inverse function `rinv_title` of `TITLE`, which is total, and therefore forces `title` to be surjective.

```
begin object class DOCUMENT
  attributes
    title : TITLE;
  ..
```

Figure 6.5: There is exactly one document for each title, and there is exactly one title for each document.

```
end object class DOCUMENT;

begin object class TITLE
  attributes
    inv_title : DOCUMENT;
    ...
  axioms
    forall t : TITLE :: title(rinv_title(t)) = t;
end object class TITLE;
```

Only the existence of such an inverse function is required. The way in which this is proved is left open, e.g. by searching or storing it.

Bijectivity

Bijectivity is the the combination of the surjectivity and injectivity properties, as in the example illustrated in figure 6.5:

```
begin object class DOCUMENT
  attributes
    title : TITLE bijection;
end object class DOCUMENT;

begin object class TITLE
  ..
end object class TITLE;
```

would be equivalent to

```
begin object class DOCUMENT
  attributes
    title : TITLE;
  ..
  axioms
    forall d1, d2 : DOCUMENT ::
      title(d1) = title(d2) -> d1 = d2;
  ..
end object class DOCUMENT;

begin object class TITLE
  attributes
    inv_title : DOCUMENT;
    ...
  axioms
    forall t : TITLE :: title(inv_title(t)) = t;
end object class TITLE;
```

6.6 Inverse attributes

An ⟨AttributeSpecification⟩ declared in the specification of class C_1, and whose codomain is a class C_2, is really a many-one relationship from C_1 to C_2. Such an attribute can be given a flag `inverse`. This means that the attribute is a surjection, and implicitly defines an additional attribute which expresses the inverse relationship.

```
begin object class obj
  attributes
    attr : result_id_sort inverse;
```

has the same effect as defining

```
begin object class result_id_sort
  attributes
    inv_attr  : SET[ obj];
  axioms
    forall x : obj , r : result_id_sort ::
        attr(x)=r <-> x In inv_attr(r);
  ...
```

6.7 Unique sets of attributes

The general form of the `keys` declaration is

```
keys
    a1, ..., an;
    b1, ..., bm;
    ...
```

where all the `a`'s and `b`'s are attributes of the class in which the `keys` section appears.

Its meaning is

```
axioms
    forall t1, t2 : S ::
        Exists(t1), Exists(t2), a1(t1)=a1(t2), ... , an(t1)=an(t2)
            -> t1 = t2;
    forall t1, t2 : S ::
        Exists(t1), Exists(t2), b1(t1)=b1(t2), ... , bn(t1)=bn(t2)
            -> t1 = t2;
    ...
```

Informally, this expresses that *in* each state of the world, the values of attributes `a1` to `an` uniquely determine an instance of the class in which the `keys` section appears. The attributes `b1` to `bm` have this same property; they are an alternative way of determining an instance of the class.

Across states of the world, these attribute combinations need not be unique. They are thus not internal nor external identifiers.

Attribute specialization rule: Attribute declarations may be repeated for subclasses, but only with a codomain that is a (strict) subsort of the original codomain.

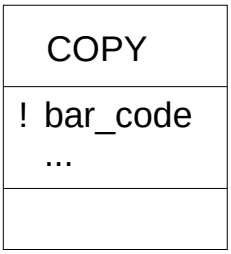

Figure 6.6: Attributes constituting an identifier are marked with a '*' in class diagrams.

6.8 Identifier section

Declaring a single attribute as `identifier` intends to express that it constitutes an external identifier, i.e. one that is visible to DBS users. What it means to be an external identifier cannot be fully specified in the formal DBS specification [15], but it can be approximated in the specification by stating that it is a key (see 6.7) and that it is fixed (see 6.4).

Stating that an attribute in LCM as `identifier`, e.g. the attribute `book_code` in

```
begin object class COPY
    attributes
        bar_code : INTEGER;
        ...
    identifiers
        bar_code;
end object class COPY;
```

expresses that `bar_code` is intended as an external identifier. Its formal meaning is equivalent to

```
begin object class COPY
    attributes
        -- Bar_code is intended as an external identifier.
        bar_code : INTEGER fixed;
        ...
    keys
        bar_code;
end object class COPY;
```

Identifiers are marked by an exclamation mark in class diagrams (figure 6.6).

6.9 Predicates specification

Within an object class specification unary predicates on the objects may be defined. These are always unary predicates with an instance of the class on which they are defined as argument. In the ⟨ObjectPredicate⟩ you cannot even specify this argument, although it is required if the object predicate is used in a ⟨PredicateTerm⟩. The syntax for defining predicates in ⟨ObjectClassBlock⟩s and ⟨RelationshipClassBlock⟩s is:

⟨Predicates⟩ ⟶ `predicates` ⟨PredicateDefinition⟩*

⟨PredicateDefinition⟩ ⟶ ⟨PredicateName⟩ *[* ⟨PredicateFlags⟩ *]* ;

⟨PredicateFlags⟩ ⟶ { ⟨PredicateFlag⟩ , }+

⟨PredicateFlag⟩ ⟶ `fixed` | ⟨PredicateInitialization⟩

⟨PredicateInitialization⟩ ⟶ `initially` ⟨PredicateTerm⟩

Example

The following predicates section, appearing in an ⟨ObjectClassBlock⟩ of object `PERSON`

```
predicates
    Married;
```

defines a predicate on persons. Applications of it should be written like `Married(p)` if used elsewhere in the specification where `p` is a variable that ranges over `PERSON`. The ⟨PredicateFlags⟩ applied to predicates have the analogous meaning as ⟨AttributeFlags⟩ with attributes.

6.10 Object existence predicate

With each ⟨ObjectClassBlock⟩ or ⟨RelationshipBlock⟩ the unary predicates `Exists` is defined implicitly on their elements. `Exists(o)` expresses whether the object or relationship `o` is viewed to exist in the current state of the world. (See Gamut [5] for a discussion of the usefulness of existence predicates in modal logic.) In addition, for ⟨ObjectClassBlock⟩s only, an additional unary predicate `Used(o)` is defined to express whether there has been a creation event for the object. In the initial world, i.e. before any event has occurred, these predicates are defined to be equivalent to `False` for all objects of the class. Creation and destruction event on the object affect the valuation of these predicate. This is described in detail in section 8.9.

6.11 Generation of default internal identifier value types

The author of a LCM specification can explicitly specify the set of values to be used as internal identifiers for the instances of a species by defining a value type with the same name as the species. If the author didn't specify the set of values to be used as internal identifiers for a species, a value type with an infinite supply of identifiers is assumed, and is given the same name as the species. In the case of unnamed species that result from defining multiple ways of partitioning a class, an internal identifier sort is defined for each of the species, e.g. four sorts for each of the intersection classes of figure 5.1 in section 5.4. You can refer to this species and its internal identifier sort by using a ⟨SortExpression⟩ containing a * to refer to denote the intersection, as explained in 5.4.

6.12 Static integrity axioms

Static integrity constraints can be expressed by including them as axioms in an axioms section. For example, we can express the integrity constraint that no person is older than 150 years by including an axiom that says so. All events that lead from an admissible state of the world to one that would violate this constraints are then blocked.

```
begin object class PERSON
  attributes
    age : NATURAL;
    ...
  axioms
```

```
    forall p : PERSON ::
      age(p) <= 150;
end object class PERSON;
```

Dependencies, even existence dependencies between objects, can be included similarly. Note that this does not result in automatic propagation of changes like cascading deletions or recomputation of derived attributes. It just states that in each state of the world the constraint should be satisfied, and simply forbids any event with an effect that would violate it.

Chapter 7

Relationships

Objects, as discussed in chapter 6 deal with attributes, predicates,events and life cycles of single objects. In order to handle n-ary predicates and n-ary attributes, that concern more than one object, we introduce relationship classes. Using them, we can express that some relationship between a fixed number of objects holds. This relationship can be assigned attributes, predicates, events and a life cycle that concerns the objects in the relationship. In addition, relationships can have their own life cycle definitions, just like objects.

The key feature that distinguishes relationship classes from other object classes is the way their instances are identified. Relationships are identified by a **composite identifier** (consisting of one identifier for each of its components), whereas ordinary objects have an atomic identifier. Thus relationships have no independent identity, but must always be identified via the identity of their components. As a corollary, we have that we cannot identify an existing relationship if one of its components is non-existent. This is discussed in more detail in section 7.2. If these assumptions about the identity and existence of a relationship are undesirable in a certain modeling context, you are probably not handling a relationship but an ordinary object instead. By turning an relationship class into an object class you give each instance its own independent identity. If desired, you could add existence constraints for objects too. For example, you can specify that an object cannot exist if an object-valued attribute does not exists. Such axioms must be supplied explicitly for objects, but they are implicit for relationships.

7.1 Relationship block

Syntax:

⟨RelationshipBlock⟩ —→ `begin relationship class` ⟨UserDefinedRelationshipName⟩
⟨RelationshipGeneralization⟩*
⟨RelationshipSpecialization⟩*
⟨Components⟩
[⟨Attributes⟩ *]*
[⟨Predicates⟩ *]*
[⟨Events⟩ *]*
[⟨Process⟩ *]*
[⟨Axioms⟩ *]*
`end relationship class` ⟨UserDefinedRelationshipName⟩ ;

⟨UserDefinedRelationshipName⟩ —→ ⟨Name⟩

The ⟨UserDefinedRelationshipName⟩s appearing in the `begin relationship class` and `end relationship class` clauses of a ⟨RelationshipBlock⟩ should be the same. This is the third instance

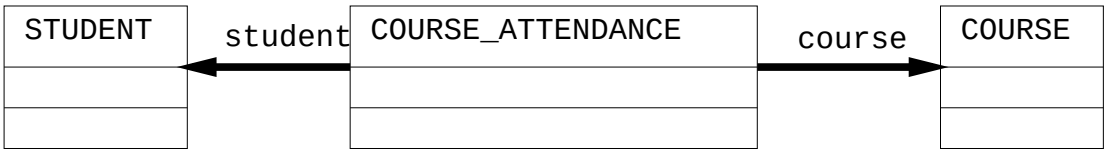

Figure 7.1: Relationship classes are shown in class diagrams as a class with fat arrows pointing to its component classes, labeled with the component names.

of the block name rule.

The occurrence o of a ⟨UserDefinedRelationshipName⟩ appearing immediately after the `begin relationship class` is a defining occurrence of that name, with $Scope(o)$ equal to the whole source text, and $Kind(o)$ = relationship, and at the same time a defining occurrence of that name with $Scope(o)$ equal to the whole source text, and $Kind(o)$ = sort. The latter is the implicitly declared sort of internal identifiers for the relationship instances. Conceptually, this sort consists of tuples of values for all components of the relationship, generated by a function called R`_id` with scope equal to the whole source text, where R is the relationship name. This function takes internal identifiers for all components of R as parameters (see section 7.4).

Example

The attendence of courses (possibly more than one) by students (possibly more than one per course) can be conveniently described using a relationship `COURSE_ATTENDANCE` below:

```
begin object class STUDENT
  ...
end object class STUDENT;

begin object class COURSE
  ..
end object class COURSE;

...

begin relationship class COURSE_ATTENDANCE
  components
    student : STUDENT;
    course : COURSE;
  attributes
    ...
end relationship class COURSE_ATTENDANCE;
```

In class diagrams, a relationship is shown by fat arrows labeled with the component names pointing from the relationship class to the component classes. Figure 7.1 shows this convention applied to the above example.

7.2 Existence dependency of relationships on their components

Relationships are existence dependent on their components. This means that a relationship `r` can only exist (i.e. the built-in predicate `Exists(r)` holds) in a world state if in the same world state all components `ci` of that relationship also have `Exists(ci)` true. Note that this does not hold

in the other direction: it is perfectly valid to have a world state in which all components of a relationship are existent, but the relationship isn't.

This is expressed as the implicit inclusion of the following axioms expressing this existence dependency:

```
axioms
    forall c1 : component_name_1, ..., cn : component_name_n ::
        Exists(R_id(c1,...,cn)) -> Exists(c1);
    ...
    forall c1 : component_name_1, ..., cn : component_name_n ::
        Exists(R_id(c1,...,cn)) -> Exists(cn);
```

7.3 Specialization

Just like object classes, relationship classes can be specialized too. The syntax of a ⟨RelationshipSpecialization⟩ occuring in a ⟨RelationshipClassBlock⟩ is similar to that given for object specialization in subsection 6.2, adapted by replacing objects with relationships. Its syntax is:

⟨RelationSpecialization⟩ ⟶ `partitioned by` { ⟨SubRelationName⟩ , }+ ;

⟨SubRelationName⟩ ⟶ ⟨UserDefinedRelationshipClassName⟩

⟨RelationshipGeneralization⟩ ⟶ `specialization of` ⟨SuperRelationshipName⟩ ;

⟨SuperRelationshipName⟩ ⟶ ⟨UserDefinedRelationshipName⟩

As a consequence, a relationship class can only contain relationship classes as specialization but not object classes.

> **Component specialization rule:** Each specialization of a relationship class inherits all components of its parent, and should not mention them again in its ⟨RelationshipClassBlock⟩, except for the case that it constrains the set of possible values for such an inherited component to a subsort.

This is similar to the attribute specialization rule.

It is perfectly reasonable for a subrelationship to have extra attributes and predicates that its parent relationship doesn't have.

This is illustrated in the following example.

Example

Consider the case of a library where borrowing of rare books is restricted to staff members only. If we would like to partition the relationship that registers borrowings, for example because different rules and attributes are applicable to them, we could express this by partitioning the relationship `BORROWING` as follows (see also figure 7.2:

```
begin object class MEMBER
  partitioned by
    STUDENT, STAFF_MEMBER;
end object class MEMBER;

begin object class STUDENT
...
```

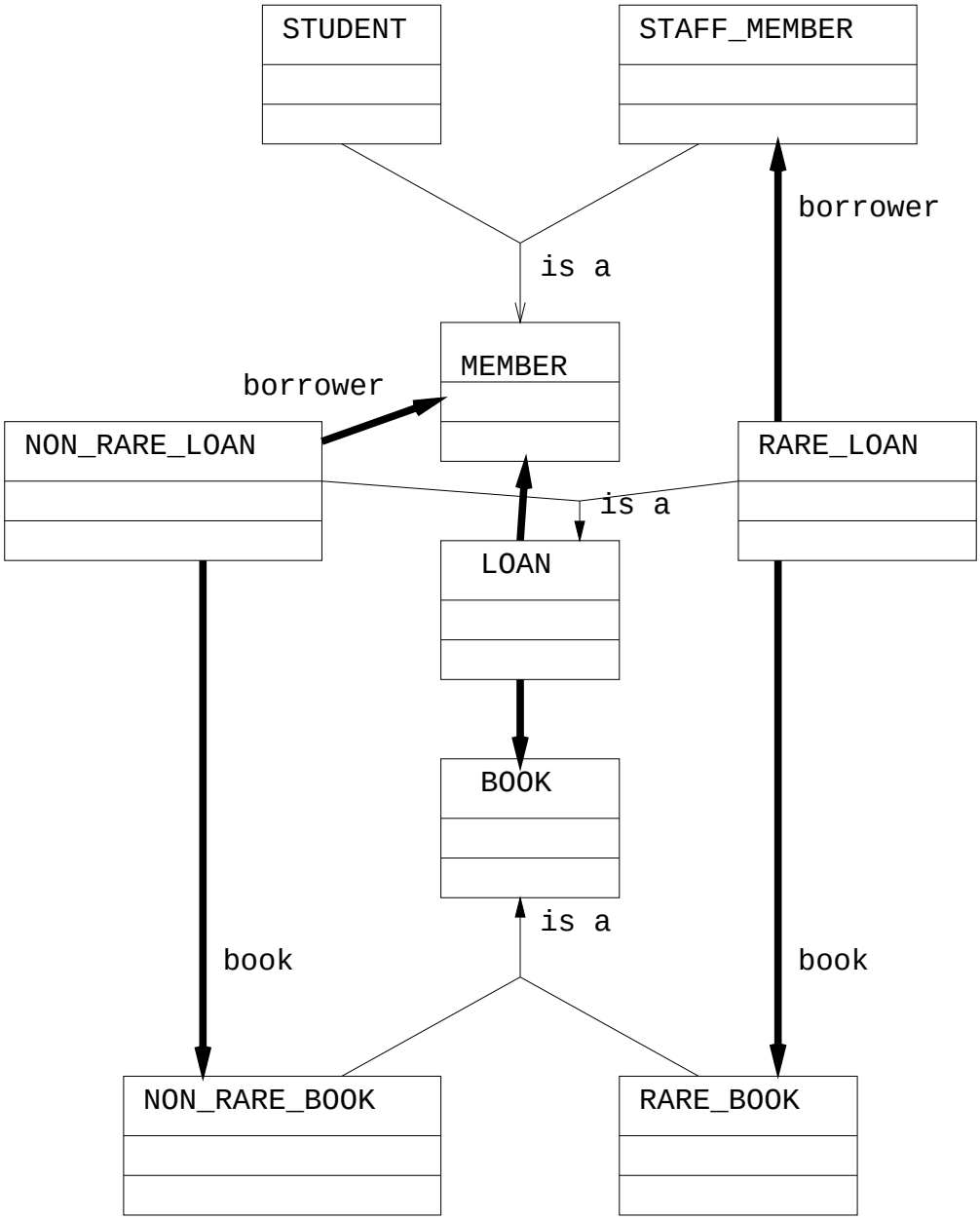

Figure 7.2: Relationship LOAN is partitioned into RARE_LOAN and NON_RARE_LOAN.

```
end object class STUDENT;

begin object class STAFF_MEMBER
...
end object class STAFF_MEMBER;

begin object class BOOK
  partitioned by
    RARE_BOOK, NON_RARE_BOOK;
end object class BOOK;

begin relationship class LOAN
  partitioned by
    RARE_LOAN, NON_RARE_LOAN;
  components
    borrower : MEMBER;
    book     : BOOK;
  attributes
    borrow_date : DATE;
...
end relationship class LOAN;

begin relationship class RARE_LOAN
  components
    borrower : STAFF_MEMBER;
    book   : RARE_BOOK;
end relationship class RARE_LOAN;

begin relationship class NON_RARE_LOAN
  components
    borrower : MEMBER;
    book : NON_RARE_BOOK;
...
end relationship class NON_RARE_LOAN;
```

7.4 Components of a relationship

Syntax:

⟨Components⟩ ⟶ `components` { ⟨ComponentSpecification⟩ }+

⟨ComponentSpecification⟩ ⟶ ⟨ComponentName⟩ : ⟨ObjectClassName⟩ ⟨AttributeFlags⟩ ;

A `components` section mentions the object classes that partitipate in the relationship. At least one ⟨ComponentName⟩s should be specified in the ⟨Components⟩ section of a ⟨RelationshipBlock⟩. The ⟨ComponentName⟩s should be names that occur as an ⟨ObjectClassName⟩ in an ⟨ObjectClassBlock⟩ or a ⟨RelationshipName⟩ in a ⟨RelationshipBlock⟩ elsewhere in the specification. Note that this includes relationship-valued components.

A LCM relationship definition of the form

```
begin relationship class R
    components
        component_name_1 : object_class_name_1 ;
```

```
            ⋮
        component_name_n : object_class_name_n ;
    ⋮
end relationship class R ;
```

is similar to an object class definition with all other sections appearing in the ⟨Relationship-ClassDefinition⟩, but which is identified by all its components instead of having its own identity.

The internal identifiers for relationship R above are defined implicitly as tuples of internal identifiers for its components. In contrast with object classes, in which case the user may supply his own specification of the data type to be used as internal identifiers for these objects, the internal identifier sorts for relationships are always defined implicitly, and are named identically to the relationship as follows:

```
begin value type R
    functions
        R_id(obj_class_1, ..., obj_class_n) : R;
        comp_1(R) : obj_class_1;
            ⋮
        comp_n(R) : obj_class_n;
    axioms
        forall r:R, c1: obj_class_1, ... , cn: obj_class_n ::
            comp_1(R_id(c1,..., cn)) = c1;
        -- similar axioms for the other components...
end value type R;
```

The constructor function is always the relationship class name with suffix `_id`; it may be used in the specification. The order in which the components appear as arguments is in increasing alphabetical order of the corresponding component names. Note that the parameters included the components that are all inherited in the case of a relationship class specialization.

Chapter 8

Local events

Local events are used to describe events that occur in the life of objects, and to associate changes in attribute and predicates values in the model with them.

The preconditions and effects of events are defined in dynamic logic. The event and life cycle definitions for objects and relationships, as well as the combination of several communicating events into transactions (see chapters 9 and 10) are modeled after process algebra[1]. We provide expressions for processes and allow recursive specifications for life cycles of objects and relationships. We will discuss this in chapter 9.

In this chapter, we focus on the declaration of local events, and on the specification of their preconditions and effects.

Sections 8.1 to 8.4 discuss event declaration, event terms and two special events `fail` and `skip`. Sections 8.5 to 8.8 discuss **precondition axioms**, which express a precondition for either the success or the failure of an event in terms of attribute and predicate values and **effect axioms**, which express the change on the local state caused by an event ocurrence. Both use dynamic logic for this purpose.

Creation and deletion events are discussed in section 8.9.

8.1 Local event definition

Local events of some object class or relationship class can be defined by including an ⟨Events⟩ section its ⟨ObjectClassBlock⟩ or ⟨RelationshipClassBlock⟩. These events are called *local* events because every occurrence of them is local to the life of an instance (an object or a relationship). This instance is called the **subject** of the event occurrences.

Different occurrences of an event can of course occur in the life of different subjects, as long as these subjects are instances of the class in whose specification the event is declared.

Syntactically, an events section looks like this:

⟨Events⟩ ⟶ `events` ⟨EventSpecification⟩*

⟨EventSpecification⟩ ⟶ ⟨EventName ⟩ ⟨EventFormalParameters⟩ *[* ⟨EventFlag⟩ *]* ;

⟨EventFormalParameters⟩ ⟶ ({ ⟨FormalEventSubject⟩ , }+ *[* ; { ⟨SortExpression⟩ , }+ *]*)

⟨EventFlag⟩ ⟶ `creation` | `deletion`

⟨FormalEventSubject⟩ ⟶ ⟨SortExpression⟩

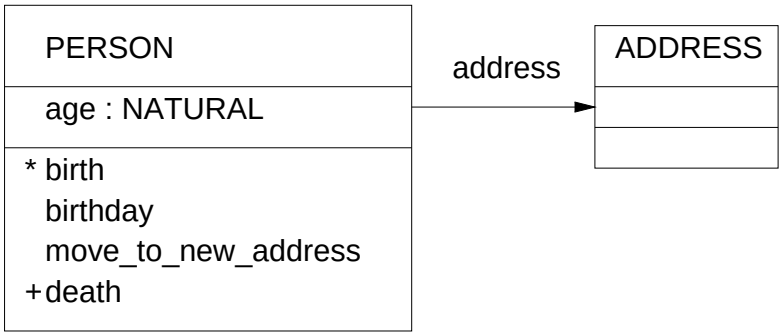

Figure 8.1: Creation and deletion events are marked in a class diagram with a '*' and '+' respectively.

The optional flags `creation` and `deletion` declare the local event to be of a special kind, viz. a creation of deletion event respectively. They are discussed in 8.9.

Local events are parametrized, using a syntax similar to the syntax for function and predicate parameters. The only difference is that it makes clear which parameter is the *subject* of a local event (i.e. the object or relationship in the life of which it takes place). After the opening parenthesis, we first mention the subject of the local event. If there are any other (i.e. non-subject) parameters, they will follow after a semicolon that separates the subject from the non-subject parameters.

In class diagrams, creation events are marked by an asterisk and deletion events by a plus, like in figure 8.1.

Examples

The next fragment of the specification of object class `PERSON` shows how four kinds of local events for PERSON are declared: `birthday`, `change_address` creation event `birth`, and deletion event `death`. (See also figure 8.1.)

```
begin object class PERSON
    attributes
        age     : NATURAL;
        address : ADDRESS;
    events
        birth(PERSON; ADDRESS)     creation;
        birthday(PERSON);
        move_to_new_address(PERSON; ADDRESS);
        death(PERSON)             deletion;
    axioms
        forall p : PERSON, ad : ADDRESS ::
            [birth(p; ad)] age(p) = 0 and address(p) = ad;

        forall p : PERSON, a : NATURAL ::
            age(p) = a -> [birthday(p)] age(p) = a + 1;

        forall p : PERSON, ad : ADDRESS ::
            [move_to_new_address(p; ad)]
                address(mover) = ad;
end object class PERSON;
```

In chapter 10 we will define transactions, which are like events, but can have more than one subject.

8.2 Refering to events

One can refer to a user-defined event, or the implicitly defined events `create` and `delete` by mentioning it, supplying actual values for its subject(s) and other parameters.

⟨EventExpression⟩ —→ ⟨UserDefinedEventName⟩ ⟨ActualEventParameters⟩

⟨UserDefinedEventName⟩ —→ ⟨Name⟩

⟨ActualEventParameters⟩ —→ (⟨ActualSubjects⟩ *[* ; ⟨ActualParameterList⟩ *]*)

⟨ActualSubjects⟩ —→ { ⟨SubjectExpression⟩ , }+

⟨SubjectExpression⟩ —→ ⟨Expression⟩

Examples of event expressions are `birth(john;ad)` and `remove(john)`.

8.3 Failure event

There is a special built-in event named `fail` which denotes eternal stagnation:

⟨EventExpression⟩ —→ `fail`

The meaning of `fail` can be described by the instances for all subjects S of the axiom

```
forall s : S :: [fail] False;
```

Literally, this means that the predicate `False` holds in all states of the world reachable by performing `fail` from the current world state. Since there are no world states in which `False` holds, this is equivalent to stating that there are no world states reachable by performing `fail`.

8.4 Skip event

The special event named `skip` denotes doing nothing. It leaves all attributes and predicates unaltered.

⟨EventExpression⟩ —→ `skip`

There are no axioms for `skip`. By the frame assumption, this means that it does not change.

8.5 Specifying the effects and preconditions of an event occurrence

The effects and admissibility of events in a particular state of the world are expressed using dynamic logic[7][8]. Two special kinds of ⟨Condition⟩s (see 5.8) are available for this purpose:

⟨Condition⟩ —→ [⟨EventExpression⟩] ⟨SimpleCondition⟩

⟨Condition⟩ —→ < ⟨EventExpression⟩ > ⟨SimpleCondition⟩

Note that the priority of the box and diamond is lower than that of predicates, thus we can often avoid writing parentheses around composite ⟨SimpleCondition⟩s following them.

An implementation will generally offer only a restricted subset from the axioms admitted by this syntax definition. The exact nature of subsets that are restricted enough to make the specification executable in practice, but are still offer a convenient way to specify what we want in practical case, is subject of current research.

In practice, separating axioms that specify the effects of events (if they occur) from axioms that constrain the admissibility of events turns out to increase readability of specifications.

Axioms of the first kind are called **effect axioms**, whereas the admissibility-constraining variants are called **precondition axioms**.

8.6 Effect axioms

Effect axioms describe the effects of a local event that are reflected in the values of the attributes and predicates of the object or relationship.

This is usually done with axioms of the form

```
forall (variables) ::
  (binding of variables to values of attributes and predicates
  before the event occurrence)
      -> [event(...)]
          (equations relating attribute values after the event occurrence
          to the variables)
```

You need only specify the changes; by the frame assumption everything of which we cannot prove that it has changed stays the same.

When specifying events on relationships, the specifier should identify the relationship using the constructor function with all components of the relationship as parameters to get the internal identifier of the relationship. This internal identifier is then passed as the subject of the event.

Example

```
begin relationship class LOAN
    components
        copy : COPY;
        member : MEMBER;
    attributes
        date_of_borrow : DATE;
    events
        borrow(LOAN; DATE) creation;
    axioms
        -- specify effect of creation event borrow:
        forall c : COPY, m : MEMBER, d : DATE, b : LOAN ::
            b = LOAN_id(c, m) ->
                [borrow(b;d)] date_of_borrow(b) = d;
end relationship class LOAN;
```

Examples

See the examples of subsection 8.7 for some effect axioms. Note that effect axioms can only affect the local state of the object or relationship in whose life the event occurs. If some real-world event has effects on the state of more than one object or relationship, then it should be modeled

Alternative form	Formal equivalent	Informal meaning
`after` e `possibly` ϕ	<e> ϕ	It is possible to execute event e and arrive in a world state where ϕ holds.
`after` e `necessarily` ϕ	[e] ϕ	If all worlds in which we arrive after executing event e ϕ holds.

Figure 8.2: Syntactically sugared forms of effect axioms.

Alternative form	Formal equivalent	Informal meaning
e `succeeds only if` ϕ	<e>`True ->` ϕ	ϕ is a necessary precondition for the success of event e (but in general not sufficient)
e `succeeds if` ϕ	ϕ `->` <e>`True`	ϕ is a sufficient precondition for the success of event e
e `fails if` ϕ	ϕ `->` [e]`False`	ϕ is a sufficient precondition for the failure of event e
e `fails only if` ϕ	[e]`False ->` ϕ	ϕ is a necessary precondition for the failure of event e

Figure 8.3: Syntactically sugared forms of precondition axioms.

as a transaction, which consists of the synchronous execution of a number of local events (see section 10).

As an alternative syntax for common forms of effect axioms, LCM provides the following syntactic sugar:

⟨Condition⟩ ⟶ `after` ⟨EventExpression⟩ ⟨Modality⟩ ⟨SimpleCondition⟩

⟨Modality⟩ ⟶ `necessarily` | `possibly`

Their meaning is summarized in figure 8.2.

8.7 Precondition axioms

LCM offers easier-to-read alternative syntax for some common forms of precondition axioms. They are summarized in figure 8.3.

The syntax is:

⟨AxiomBody⟩ ⟶ ⟨EventExpression⟩ ⟨FailOrSucceed⟩ ⟨Cond⟩ ⟨SimpleCondition⟩

⟨FailOrSucceed⟩ ⟶ `fails` | `succeeds`

⟨Cond⟩ ⟶ `if` | `only if`

Note that expressions like e `succeeds if` ϕ are misleading in the sense that in fact the sufficient precondition for success is not ϕ, but in general the conjunction of ϕ and all global integrity constraints!

Examples

```
begin object class LIBRARY_MEMBER
    attributes
        no_of_books_borrowed : NATURAL initially 0;
        ...
    events
        -- A library member borrows a book.
        borrow(LIBRARY_MEMBER);
        -- A library member returns a book.
        return(LIBRARY_MEMBER);
    axioms
        -- Effect axiom: increase amount of books borrowed
        -- each time a book is borrowed.
        forall m : LIBRARY_MEMBER, book_count : NATURAL ::
            no_of_books_borrowed(m) = book_count ->
                [borrow(m)] no_of_books_borrowed(m) = book_count + 1;


        -- Precondition axiom: a member may have borrowed at most three
        -- books at any time.
        borrow(m) succeeds only if no_of_books_borrowed(m)  < 3;
        ...
end object class LIBRARY_MEMBER;
```

Instead of the precondition axiom above, we could also have used

```
        borrow(m) fails if no_of_books_borrowed(m) >= 3;
```

If we want, we can still use the dynamic logic form:

```
        <borrow(m)>True -> no_of_books_borrowed(m) < 3;
```

Note that in this example we could have handled this constraint better by turning it into a local static constraint, which should apply to all states of a `LIBRARY_MEMBER`, and thus not only in states immediately after a `borrow` event:

```
        no_of_borrowed_books(m) <= 3;
```

Ideally, a tool could prove such invariants from the precondition and effect axioms and the specification of the initial state, or generate expanded precondition axioms which together ensure that such an invariant cannot be violated. Precondition axioms for an event may refer to the state of other objects, but the event can only have effect on the local state of the object in the life of which the event takes place.

8.8 Implicit existence precondition

All local events except creation events (user-defined as well as default) have as implicit precondition for success that their subject `s` exists. This can be viewed as the automatic inclusion of a precondition axiom of the form

```
forall s : ...subject,... ::
    <nce(s;...)>True -> Exists(s);
```

for each non-creation event `nce`.

See subsection 8.9 for implicit preconditions for creation events.

8.9 Creation and deletion events

Creation events and deletion events are events that have an effect on the existence of an object or relationship. Creation and deletion events are only allowed for **species**, i.e. smallest classes in the taxonomy. Due to their special nature, and their intimate connection with relationships and the built-in predicates `Exists` and `Used` (see section 5.10), they get a special treatment. This avoids clerical work and thus reduces the possibility of getting hard-to-find bugs introduced by effect and precondition axioms that don't take the special nature of these events into account correctly.

8.9.1 The meaning of creation events

Each creation event has the requirement that its subject doesn't already exist as a precondition for success, so for all creation events `cr`, we have implicit axioms saying

```
forall s : ...subject,... ::
    Exists(s) -> [cr(s;...)]False;
```

In addition, to avoid confusion due to reuse of internal identifiers, we have

```
forall s : ...subject...,...::
    Used(s) -> [cr(s;...)]False;
```

The effect of a creation event, if it succeeds, is always that the object comes into existence

```
forall s : ...subject...,...::
    [cr(s;...)] Exists(s);
```

and the use of the internal identifier is registered:

```
forall s : ...subject...,...::
    [cr(s;...)] Used(s);
```

An attempt to create a relationship while one of its components is nonexistent will fail due to the global existence dependency of relationships in combination with these axioms. This is discussed in detail in section 7.2.

Since the smallest subclass of an object or relationship must be known at creation/deletion time, you can only define creation and deletion events for species. See section 5.4 for ways to refer to species that don't have a explicit user-defined name.

User-defined creation and deletion events are mentioned in an ⟨EventSpecification⟩ and given a meaning by user-supplied effect axioms for them. If the user does not supply creation or deletion events for a species, LCM automatically supplies standard creation and deletion events. Such **default creation** and **deletion events** have `create` and `delete` as their names, respectively, and are discussed in subsections 8.9.2 and 8.9.4. The user may declare an event called `create`, but then it must be a creation event (and marked as such by the `creation` flag). Analogously, the user may declare an event called `delete`, but then it must be a deletion event (and marked as such by the `deletion` flag).

8.9.2 Default creation event

If the user has marked no events for a species as `creation`, a default creation event `create` is defined implicitly. This default creation event initializes all attributes to a value which comes from an initial value specification, or, if this is not present, from a parameter of `create`. These parameters should appear in increasing lexicographic ordering of the names of the uninitialized attributes, taking the ordering in ASCII as basis.

The next example will show the idea:

Example

Consider the following object class specification (where `ATTR1` to `ATTRn` are value types defined elsewhere, and `i2` is an initial value expression of type `ATTR2`; we assume that `MyClass` is a species) :

```
begin object class MyClass
    attributes
        a1 : ATTR1;
        a2 : ATTR2     initially i2;
        ...
        an : ATTRn;
end object class MyClass;
```

As this species has no user-supplied creation event, a default creation event named `create` is supplied, which has the effect as if the user would instead have written:

```
begin object class MyClass
    attributes
        a1 : ATTR1;
        a2 : ATTR2                                  initially i2;
        a3 : ATTR3;
        ...
        an : ATTRn;
    events
        -- Default creation event:
        -- Note the absence of a parameter for a2:
        -- the  value i2 is used as initialization of a2.

        -- Arguments to create after the semicolon are ordered
        -- based on the ASCII ordering on the names of the attributes
        -- for which they supply a value.
        create(MyClass; ATTR1, ATTR3, ..., ATTRn)      creation;
    axioms
        -- Allow creation of instances of MyClass.
        forall s : MyClass, b1 : ATTR1, ..., bn : ATTRn ::
            -- Precondition of nonexistence of s, and
            -- Postcondition of existence of s are implicit!
            [create(s; b1, b3, ..., bn)]
                a1(s)=b1, a2(s)=i2, ..., an(s)=bn;
end object class MyClass;
```

8.9.3 Specifying initial values of attributes and predicates in the default creation event

An ⟨Initialization⟩ flag of an attribute declaration specifies that on creation of an instance of this class the attribute or predicate for which it is given gets the given value. Note that this applies to all creation events, user-supplied (possibly more than one!) as well to as default creation events.

Syntax of initial value specification for attributes and predicates:

⟨Initialization⟩ ⟶ `initially` ⟨InitialValueExpression⟩

⟨InitialValueExpression⟩ ⟶ ⟨Expression⟩

The ⟨InitialValueExpression⟩ must be a constant expression, i.e. it involves only functions, constant predicates and constants, but not attributes or predicates that depend on the world state.

The type of the ⟨InitialValueExpression⟩ must be a (non-strict) subsort of the attribute type in case of an initialization for an attribute, or must be one of the built-in predicates `True` or `False`.

8.9.4 Default deletion events

Analogously to creation events, a default deletion event named `delete` is supplied if there is no user-supplied deletion event. This event has only the internal identifier of the object or relationship to be deleted as argument. The effect of a deletion event on the built-in predicate `Exists` is described by implicit axioms for a deletion event `del`. It doesn't change the built-in predicate `Used`.

```
forall s: ...subject...,...::
    <del(s;...)> -> Exists(s);
forall s: ...subject...,...::
    [del(s;...)] not Exists(s);
```

Chapter 9

Life cycles

Life cycle definitions describe regularities in the various events that occur during the life of an object or relationship. They might, for example, constrain the order in which they can occur. Note that life cycle specifications express constraints on the possibility of the occurrence of an event, just as preconditions do. Life cycles can be specified using process operators to combine simple events into processes. The sublanguage for specifying the processes that form the life cycles of objects and relationships is based on process algebra. The intended semantics of a recursive process definition is a process graph model [1].

The life cycle of an object or relationship is specified by including a ⟨LifeCycle⟩ section in its block. If no life cycle is specified, a simple default life cycle definition is assumed; see section 9.3.

Example

As an example of a life cycle definition, consider this one from the domain of coffee machines, which has unsurpassed popularity in the process algebra texts.

```
begin object class VENDING_MACHINE
  events
    insert_coin(VENDING_MACHINE);
    give_coffee(VENDING_MACHINE);
  life cycle
    forall v : VENDING_MACHINE ::
      VENDING_MACHINE(v) = create(v) . IN_OPERATION(v);
    forall v : VENDING_MACHINE ::
      IN_OPERATION(v) = insert_coin(v) . give_coffee(v) . IN_OPERATION(v);
end object class VENDING_MACHINE;
```

This definition enforces a certain protocol on the instances of `VENDING_MACHINE`: it specifies a required ordering in time of the occurences the events of the vending machine. For example, an `insert_coin` event can only occur in the future of the life of a particular `VENDING_MACHINE` if a `create` event has occured in its past. Also, the machine will only perform a `give_coffee` event, if immediately its most recent event was `insert_coin`.

The process described by these two ⟨ProcessDefinition⟩s can be visualized as a process graph, as shown in figure 9.1.

9.1 Life cycle definition

Syntax:

⟨LifeCycle⟩ ⟶ `life cycle` ⟨ProcessDefinition⟩*

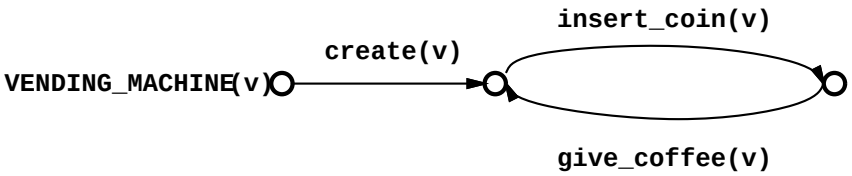

Figure 9.1: A process graph of the life cycle specification of a vending machine.

⟨BinaryInfixProcessOperator⟩	Meaning
.	Sequential composition: a.b means first a occurs, then b
||	Parallel composition
+	Choice: a+b means a or b occurs, but it is not said which
&	Synchronous communication

Figure 9.2: Binary infix process operators, showed in order of decreasing priority. All associate from the left to right.

⟨ProcessDefinition⟩ ⟶ ⟨Quantification⟩ : : ⟨ProcessName⟩ *[* ⟨FormalProcessParameters⟩ *]*
= ⟨ProcessExpression⟩ ;

⟨ProcessExpression⟩ ⟶ ⟨FactorProcessExpression⟩

⟨ProcessExpression⟩ ⟶ ⟨FactorProcessExpression⟩ ⟨BinaryInfixProcessOperator⟩ ⟨FactorProcessExpression⟩

⟨FactorProcessExpression⟩ ⟶ ⟨EventExpression⟩

⟨FactorProcessExpression⟩ ⟶ ⟨ProcessName⟩ *[* ⟨ActualProcessParameters⟩ *]*

⟨FactorProcessExpression⟩ ⟶ (⟨ProcessExpression⟩)

⟨ActualProcessParameter⟩ ⟶ ⟨ConstantExpression⟩ | ⟨BoundVariable⟩

⟨BinaryInfixProcessOperator⟩ ⟶ . | + | ||

You always have to write an explicit `forall` in a process definition, which binds the subjects and/or parameters used in the ⟨ProcessParameters⟩ and ⟨ProcessExpression⟩.

The ⟨BinaryInfixProcessOperator⟩s are shown in figure 9.2.

The scope of a defining occurrence of a variable name in the ⟨Quantification⟩ of a ⟨Process-Definition⟩ is that whole ⟨ProcessDefinition⟩; note that this differs from the approach of implicit scoping chosen in e.g. CCS [9].

The occurrence o of a name as ⟨ProcessName⟩ of a ⟨ProcessDefinition⟩ is a defining occurrence of that name, with the whole LCM source text as its scope, and $Kind(o) =$ process.

The connection between the process definitions and the lifecycle of an object or relationship is made by including a ⟨ProcessDefinition⟩ with the name of the object class or relationship class as the ⟨ProcessName⟩. This is understood to mean the life cycle processes of their instances.

This process has the subject (i.e. the internal identifier of the object or relationship) as first argument, separated by a semicolon from the other arguments. The subject specification may be

omitted; in this case a subject of the object or relationship class in which the ⟨ProcessDefinition⟩ occurs is assumed, just as with event parameters.

If the event has other parameters apart from its subject, they should be specified in each ⟨EventExpression⟩. The optionality of the parameters in the EBNF definition is only intended to cover the implicit subjects, and the event without further parameters.

If a ⟨ProcessDefinition⟩ defining the life cycle for an object or relationship is included, it should explicitly include its creation and deletion events, even if they the default creation and deletion events are assumed.

9.2 Guardedness of lifecycle specifications

In order to be sure that the ⟨FactorProcessExpression⟩s together uniquely determine a process, all ⟨FactorProcessExpression⟩s are required to be **completely guarded**. An occurrence of a ⟨BoundVariable⟩ in a ⟨ProcessExpression⟩ is called guarded if this ⟨ProcessExpression⟩ has a subexpression of the form a `.` t where a is an ⟨EventExpression⟩ that is not `fail` or `skip` and t is a term containing this occurrence of the ⟨BoundVariable⟩. We say that a ⟨FactorProcessExpression⟩ is completely guarded if each occurrence of a ⟨BoundVariable⟩ in it is guarded.

Note that we demand a life cycle specification to be *completely* guarded instead of just guarded; this stronger requirement is imposed in order to make this test easily decidable by looking at the specification, whereas requiring only guardedness could need complicated rewriting using ACP axioms in order to decide whether it happens to be equivalent to some completely guarded specification, for example by rewriting $X = Y.X$ and $Y = a.Y$ to $X = a.Y.X$ and $Y = a.Y$

```
life cycle
  forall v :: VENDING_MACHINE ::
    VENDING_MACHINE(v) = COINS(v) . give_coffee(v) . VENDING_MACHINE(v);
  forall v :: VENDING_MACHINE ::
    COINS(v) = insert_coin(v) . COINS(v);
```

to the completely guarded equivalent

```
life cycle
  forall v :: VENDING_MACHINE ::
    VENDING_MACHINE(v) = insert_coin(v) . MORE_COINS(v) . VENDING_MACHINE(v);
  forall v :: VENDING_MACHINE ::
    MORE_COINS(v) = insert_coin(v) . MORE_COINS(v) + skip;
```

Example

Both ⟨ProcessExpression⟩s in the `VENDING_MACHINE` example above are completely guarded. They could be be derived from

```
life cycle
   VENDING_MACHINE = VENDING_MACHINE . insert_coin . give_coffee;
```

which is guarded but not completely guarded,

9.3 Default life cycle

If no ⟨ProcessDefinition⟩ with the ⟨ProcessName⟩ equal to the name of a species appears anywhere in the specification, a default life cycle definition is assumed, which has the form

```
forall x : OBJ::
  OBJ(x) = _OBJ_CREATION(x) . _OBJ_LIFE(x);
forall x : OBJ::
```

```
  _OBJ_CREATION(x) = c1(x) + c2(x) + ... + cn(x);
forall x : OBJ::
  _OBJ_LIFE(x) = (e1(x) + e2(x) + ... + en(x)) . _OBJ_LIFE(x) +
                 d1(x) + d2(x) + ... + dn(x);
```

where `c1` to `cn` are all creation events for the species (default or user-supplied), `d1` to `dn` are all deletion events for the species (again, default or user-supplied) and `e1` to `en` are all other events declared for the species.

Chapter 10

DBS transactions

The specification of DBS transactions in a ⟨ServiceBlock⟩ serves two purposes. First, it allows the definitions of events with an effect that is not limited to a single object or relationship (this is in contrast with the local events (chapter 8) which we discussed until now).

Second, it defines the boundary between the DBS and its environment, by enumerating exhaustively the transactions that the DBS can perform. This means that all communications between objects are also communications between the DBS and its environment. A **transaction** in this context is an atomic state transition of the DBS: it does take place as a whole or doesn't take place at all.

These two ideas are interlinked by defining a DBS transaction to be a special kind of event (also known as global event) which consist of the synchronous occurrence of one or more local events. Each of these local events takes place in different object or relationships.

> **Local event hiding rule**: All events except transactions are hidden.

Every DBS transaction consists of one or more local events (except skip or fail).

Every communication between objects is a DBS transaction.

All communications are global: if t is a communication of n events, then these are local to n different objects.

To express interactions of objects by communications in order to provide services in the domain of interest, a ⟨ServiceBlock⟩ is used.

10.1 Service block

Syntax:

⟨ServiceBlock⟩ ⟶ `begin service` ⟨ServiceName⟩
[⟨Transactions⟩ *]*
[⟨Decompositions⟩ *]*
`end service` ⟨ServiceName⟩;

⟨Transactions⟩ ⟶ `transactions` ⟨Transaction⟩*

⟨Transaction⟩ ⟶ ⟨TransactionName⟩ *[* ⟨EventFormalParameters⟩ *]* ;

⟨Decompositions⟩ ⟶ `decompositions` ⟨Decomposition⟩*

	transfer	open_a_new_account
ACCOUNT	withdraw deposit	open
PERSON		open_an_account

Figure 10.1: A simple transaction decomposition table showing the transactions and object classes of the example.

⟨Decomposition⟩ ⟶ ⟨Quantification⟩ ::
⟨TransactionName⟩ *[* ⟨ActualEventParameters⟩ *]* =
⟨Communication⟩;

⟨Communication⟩ ⟶ { ⟨CommunicationEvent⟩ & }+

⟨CommunicationEvent⟩ ⟶ *[* ⟨SubjectClassName⟩ . *]* ⟨EventName⟩ *[* ⟨ActualEventParameters⟩ *]*

The example below illustrates that the ⟨SubjectClassName⟩ in a ⟨CommunicationEvent⟩ is only intended to aid the reader. It can always be omitted, because the event should be unambiguous without it too (overloading of event names can be resolved by examining the sorts of the parameters).

The decomposition of the transactions defined here can be shown informally in a **transaction decomposition table** (figure 10.1).

The ⟨ServiceName⟩ appearing after the `begin services` should be the same as the⟨ServiceName⟩ after the `end services`. This is an instance of the block name rule. The ⟨ServiceName⟩ should not be declared elsewhere with global scope.

In a ⟨ServiceBlock⟩ `begin service` S ... `end service` S; the occurrence of a name immediately after `begin service` is a defining occurrence of that name, which the whole LCM source text as scope and of kind service.

Each ⟨TransactionName⟩ occurring in a ⟨Decomposition⟩ in a ⟨ServiceBlock⟩ should be declared as a ⟨Transaction⟩ in a ⟨Transactions⟩ of that block and vice versa.

The scope of a ⟨TransactionName⟩ declared by mentioning it in a ⟨Transaction⟩ in a ⟨ServiceBlock⟩ is global. There must not be another declaration of the same ⟨TransactionName⟩ with global scope.

Example

The next example show the connection between the various parts of the specification.

```
begin object class PERSON
  events
    open_an_account(PERSON);
end object class PERSON;

begin object class ACCOUNT
  attributes
    owner : PERSON;
    balance : INTEGER;              initially 0;
  events
    open(ACCOUNT; PERSON)           creation;
    withdraw(ACCOUNT; NATURAL);
    deposit(ACCOUNT; NATURAL);
  axioms
    -- precondition and effect axioms for open, withdraw and deposit...
```

```
end object class ACCOUNT;

begin services MoneyTraffic
  transactions
    transfer(ACCOUNT,  -- source account
             ACCOUNT;  -- destination account
             NATURAL); -- amount
  decompositions
    forall src, dst : ACCOUNT; amt : NATURAL ::
      transfer(src, dst; amt) = withdraw(src; amt) & deposit(dst; amt);
end services MoneyTrafic;

begin services AccountHandling
  transactions
    open_a_new_account(PERSON, ACCOUNT);
  decompositions
    forall p : PERSON, a : ACCOUNT:
      open_a_new_account(p,a) = ACCOUNT.open(a;p) & PERSON.open_an_account(p);
end services AccountHandling;
```

Chapter 11

Conclusions

LCM is still under development. Version 3.0 contains a number of features which must be considered as part of an experiment.

Modularity . One experiment is to see whether we can get away with having only two modularization constructs: the class specification and the service specification. Both constructs are orthogonal and only one (the class) is formally defined. Service specifications have only informal meaning, and corresponds with the grouping of DBS transactions in the function decomposition tree. For the user, this is (or ought to be) a significant and meaningful grouping.

The modularization construct that we dispensed with is the module construct, that contaisn one or more class specifications. Modules are usually related by import relationships. One reason for dispensing with this construct is that with the advent of hypertext systems, partitioning of a linear text into chunks called modules may become obsolete; this is a hypothesis to be tested by actually using LCM 3.0 in group projects.

Quantification. Currently, all axioms and all life cycle definitions are prefixed with explicit quantifications. This produxces a lot of visual clutter, not to say noise. Dropping all these quantification could cause ambiguity in connection with overloading of function names, attribute names or event names; moving the to one place can only be done if all quantifications are universal and then still causes a problem wih the distance between the declaration and the use of a variable. We are not satisfied with the current syntax, but defer a more satisfactory solution until after we have gained more experience with working with cmsl 3.0.

Tools. There is a LCM 3.0 mode of emacs and parser for LCM 3.0, generated by LLgen [4]. The parser can be called from the editor to syntax-check the specification. A visual editor for class diagrams is under development [3]. Eventually, a collection of diagram editors should be able to cooperate with an intelligent text editor to help writing LCM specifications easily.

A special tool to be developed is a theorem-prover for LCM. One of the tasks of a theorem prover would be to search for paths from one database state to another. Other tasks would include reasoning about invariants, reasoning about updates (cascaded updates) etc.

Additions to LCM. Additions planned for LCM include the ability to define derived attributes, dynamic subclasses and roles [16], more general cardinality constraints, and facilities for real time system specification and well as for deontic constraints. These additions will however only be added after their formal semantics and logic is known. Current research includes the design of intended semantics for LCM specifications that allow an operational semantics as well as resolution of queries by a theorem prover.

Appendix A

Collected syntax of LCM

Lexical item identification rule The next lexical item starting at some position (if any) is the longest possible substring of the specification text starting at that position that constitutes a valid ⟨LexicalElement⟩.

Unique definition constraint: We require that for every occurrence of a name in a specification there is exactly one defining occurrence for that name.

Block name rule: the names occurring in the opening and closing phrases of a block must be identical.

Non-circularity constraint: There doesn't exist a (possibly empty) sequence of ⟨SubSortExpression⟩s $S_1, \ldots, S_n$ such that `specialized by` S_2`,...` appears in the ⟨ValueBlock⟩ for S_1, and `specialized by` S_3`,...` appears in the ⟨Valueblock⟩ for S_2, and so on to `specialized by` S_n`,...` appearing in the ⟨ValueBlock⟩ of S_{n-1}. Note that this also excludes the immediate case of a ⟨ValueBlock⟩ for `S` containing `specialized by S;`.

Attribute specialization rule: Attribute declarations may be repeated for subclasses, but only with a codomain that is a (strict) subsort of the original codomain.

Component specialization rule: Each specialization of a relationship class inherits all components of its parent, and should not mention them again in its ⟨RelationshipClassBlock⟩, except for the case that it constrains the set of possible values for such an inherited component to a subsort.

Single attribute initialization rule: Only one initialization as attribute flag is allowed in an ⟨AttributeSpecification⟩, even if the values match.

Local event hiding rule: All events except transactions are hidden.

⟨ActualEventParameters⟩ —→ (⟨ActualSubjects⟩ *[* ; ⟨ActualParameterList⟩ *]*)

⟨ActualParameterList⟩ —→ ({ ⟨Expression⟩ , }+)

⟨ActualProcessParameter⟩ —→ ⟨ConstantExpression⟩ | ⟨BoundVariable⟩

⟨ActualSubjects⟩ —→ { ⟨SubjectExpression⟩ , }+

⟨ActualValueBlockParameterList⟩ —→ [{ ⟨ActualValueBlockParameter⟩ , } +]

⟨ActualValueBlockParameter⟩ ⟶ ⟨SortExpression⟩ |
`function` ⟨PrefixForm⟩ : ⟨SortExpression⟩ |
`predicate` ⟨PrefixForm⟩

⟨AttributeFlags⟩ ⟶ { ⟨AttributeFlag⟩ , }*

⟨AttributeFlag⟩ ⟶ ⟨Initialization⟩

⟨AttributeFlag⟩ ⟶ `fixed`

⟨AttributeFlag⟩ ⟶ `injection` | `surjection` | `bijection`

⟨AttributeFlag⟩ ⟶ `inverse`

⟨AttributeSpecification⟩ ⟶ ⟨AttributeName⟩ : ⟨SortExpression⟩ ⟨AttributeFlags⟩ ;

⟨Attributes⟩ ⟶ `attributes` { ⟨AttributeSpecification⟩ }* { ⟨UniquenessConstraint⟩ }*

⟨AxiomBody⟩ ⟶ ⟨EventExpression⟩ ⟨FailOrSucceed⟩ ⟨Cond⟩ ⟨SimpleCondition⟩

⟨AxiomBody⟩ ⟶ *[* ⟨Condition⟩ -> *]* ⟨Condition⟩

⟨AxiomBody⟩ ⟶ *[* ⟨Condition⟩ <-> *]* ⟨Condition⟩

⟨Axioms⟩ ⟶ `axioms` ⟨Axiom⟩*

⟨Axiom⟩ ⟶ *[* ⟨Quantification⟩ ::*]* ⟨AxiomBody⟩ ;

⟨BinaryInfixProcessOperator⟩ ⟶ . | + | ||

⟨CharacterConstant⟩ ⟶ ' ⟨CharacterSpecification⟩ '

⟨CharacterSpecification⟩ ⟶ any single printable character | ⟨SpecialCharacterConstant⟩

⟨Comment⟩ ⟶ -- ⟨NonNewLineCharacter⟩*

⟨CommunicationEvent⟩ ⟶ *[* ⟨SubjectClassName⟩ . *]* ⟨EventName⟩ *[* ⟨ActualEventParameters⟩ *]*

⟨Communication⟩ ⟶ { ⟨CommunicationEvent⟩ & }+

⟨ComponentSpecification⟩ ⟶ ⟨ComponentName⟩: ⟨ObjectClassName⟩ ⟨AttributeFlags⟩ ;

⟨Components⟩ ⟶ `components` { ⟨ComponentSpecification⟩ }+

⟨Condition⟩ ⟶ ⟨SimpleCondition⟩

⟨Condition⟩ ⟶ < ⟨EventExpression⟩ > ⟨SimpleCondition⟩

⟨Condition⟩ ⟶ [⟨EventExpression⟩] ⟨SimpleCondition⟩

⟨Condition⟩ ⟶ `after` ⟨EventExpression⟩⟨Modality⟩ ⟨SimpleCondition⟩

⟨Cond⟩ ⟶ `if` | `only if`

⟨Constant⟩ ⟶ ⟨ZeroConstant⟩ |⟨PosIntConstant⟩ | ⟨RationalConstant⟩ |⟨CharacterConstant⟩ | ⟨StringConstant⟩

⟨DecimalDigit⟩ ⟶ 0 | ⟨NonZeroDecimalDigit⟩

⟨Decompositions⟩ ⟶ `decompositions` ⟨Decomposition⟩*

⟨Decomposition⟩ ⟶ ⟨Quantification⟩ ::
⟨TransactionName⟩ *[* ⟨ActualEventParameters⟩ *]* =
⟨Communication⟩;

⟨EventExpression⟩ ⟶ ⟨UserDefinedEventName⟩ ⟨ActualEventParameters⟩

⟨EventExpression⟩ ⟶ `fail`

⟨EventExpression⟩ ⟶ `skip`

⟨EventFlag⟩ ⟶ `creation` | `deletion`

⟨EventFormalParameters⟩ ⟶ ({⟨FormalEventSubject⟩ , }+ *[* ; {⟨SortExpression⟩ , }+ *]*)

⟨EventSpecification⟩ ⟶ ⟨EventName ⟩ ⟨EventFormalParameters⟩ *[* ⟨EventFlag⟩ *]* ;

⟨Events⟩ ⟶ `events` ⟨EventSpecification⟩*

⟨Expression⟩ ⟶ ⟨SimpleExpression⟩ { ⟨BinaryInfixOperator⟩ ⟨SimpleExpression⟩ }*

⟨FactorProcessExpression⟩ ⟶ ⟨EventExpression⟩

⟨FactorProcessExpression⟩ ⟶ ⟨ProcessName⟩ *[* ⟨ActualProcessParameters⟩ *]*

⟨FactorProcessExpression⟩ ⟶ (⟨ProcessExpression⟩)

⟨FailOrSucceed⟩ ⟶ `fails` | `succeeds`

⟨FormalEventSubject⟩ ⟶ ⟨SortExpression⟩

⟨FormalParameterList⟩ ⟶ ({ ⟨FormalParameter⟩ , }+)

⟨FormalParameter⟩ ⟶ ⟨SortExpression⟩

⟨FunctionDeclaration⟩ ⟶ ⟨PrefixForm⟩ : ⟨SortExpression⟩ ;

⟨Functions⟩ ⟶ `functions` ⟨FunctionDeclaration⟩*

⟨IdentifierConstraint⟩ ⟶ ⟨AttributeName⟩ ;

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ ⟨InfixPredicate⟩ ⟨Expression⟩

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ != ⟨Expression⟩

⟨InfixPredicateApplication⟩ ⟶ ⟨Expression⟩ = ⟨Expression⟩

⟨InitialValueExpression⟩ ⟶ ⟨Expression⟩

⟨Initialization⟩ ⟶ `initially` ⟨InitialValueExpression⟩

⟨KeyConstraint⟩ ⟶ { ⟨AttributeName⟩ , }+ ;

⟨LCM-SpecificationText⟩ ⟶ ⟨LexicalElement⟩*

⟨LCM-Specification⟩ ⟶ ⟨ValueBlock⟩* { ⟨ObjRelBlock⟩ }* ⟨ServiceBlock⟩*

⟨Letter⟩ ⟶ `a` | `b` | `c` | ··· | `z` | `A` | `B` | `C` | ··· | `Z`

⟨LexicalElement⟩ ⟶ ⟨ReservedWord⟩ | ⟨Name⟩ | ⟨Constant⟩ | ⟨WhiteSpace⟩

⟨LifeCycle⟩ ⟶ `life cycle` ⟨ProcessDefinition⟩*

⟨Modality⟩ ⟶ `necessarily` | `possibly`

⟨NameContinuationCharacter⟩ ⟶ ⟨Letter⟩ | ⟨DecimalDigit⟩ | _

⟨NameStartCharacter⟩ ⟶ ⟨Letter⟩ | _

⟨Name⟩ ⟶ ⟨NameStartCharacter⟩ ⟨NameContinuationCharacter⟩*

⟨NonZeroDecimalDigit⟩ ⟶ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨NumberConstant⟩ ⟶ ⟨ZeroConstant⟩ | ⟨PosIntConstant⟩ | ⟨RationalConstant⟩

⟨ObjRelBlock⟩ ⟶ ⟨ObjectClassBlock⟩ | ⟨RelationshipClassBlock⟩

⟨ObjectClassBlock⟩ ⟶ `begin object class` ⟨UserDefinedObjectClassName⟩
⟨ObjectGeneralization⟩*
⟨ObjectSpecialization⟩*
[⟨Attributes⟩ *]*
[⟨Predicates⟩ *]*
[⟨Events⟩ *]*
[⟨LifeCycle⟩ *]*
[⟨Axioms⟩ *]*
`end object class` ⟨UserDefinedObjectClassName⟩ ;

⟨ObjectGeneralization⟩ ⟶ `specialization of{` ⟨SuperClassName⟩ `,` `}+` ;

⟨ObjectSpecialization⟩ ⟶ `partitioned by` { ⟨ObjectSubClassName⟩ `,` }+ ;

⟨ObjectSubClassName⟩ ⟶ ⟨UserDefinedObjectClassName⟩

⟨OctalDigits⟩ ⟶ ⟨OctalDigit⟩ | ⟨OctalDigit⟩ ⟨OctalDigit⟩ | ⟨OctalDigit⟩ ⟨OctalDigit⟩ ⟨OctalDigit⟩

⟨OctalDigit⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨PosIntConstant⟩ ⟶ ⟨Zero⟩+ ⟨NonZeroDecimalDigit⟩ ⟨DecimalDigit⟩*

⟨PredefinedSortName⟩ ⟶ `ZERO` | `POSINT` | `NEGINT` |`INATURAL` | `NZINTEGER` | `NATURAL` | `INTEGER` |
`NZRATIONAL` | `RATIONAL` | `DATE` |`CHARACTER` | `EMPTY` | `STRING` | `SET` |
`BAG` |`SEQUENCE` | `DATE`

⟨PredicateApplication⟩ ⟶ ⟨PrefixPredicateApplication⟩ | ⟨InfixPredicateApplication⟩

⟨PredicateDefinition⟩ ⟶ ⟨PredicateName⟩ *[*⟨PredicateFlags⟩ *]* ;

⟨PredicateFactor⟩ ⟶ { ⟨PredicateLiteral⟩ `and` }+

⟨PredicateFlags⟩ ⟶ { ⟨PredicateFlag⟩ `,` }+

⟨PredicateFlag⟩ ⟶ `fixed` | ⟨PredicateInitialization⟩

⟨PredicateInitialization⟩ ⟶ `initially` ⟨PredicateTerm⟩

⟨PredicateLiteral⟩ ⟶ *[* `not` *]* ⟨PredicateApplication⟩

⟨PredicateTerm⟩ ⟶ { ⟨PredicateFactor⟩ `or` }+

⟨Predicates⟩ ⟶ `predicates` ⟨PredicateDefinition⟩*

⟨PrefixForm⟩ ⟶ ⟨Name⟩ *[* ⟨FormalParameterList⟩ *]*

⟨PrefixFunctionApplication⟩ ⟶ ⟨PrefixFunctionName⟩ *[* ⟨ActualParameterList⟩ *]*

⟨PrefixPredicateApplication⟩ ⟶ ⟨PrefixPredicateName⟩ *[* ⟨ActualParameterList⟩ *]*

⟨ProcessDefinition⟩ ⟶ ⟨Quantification⟩ `::` ⟨ProcessName⟩ *[* ⟨FormalProcessParameters⟩ *]*
= ⟨ProcessExpression⟩ ;

⟨ProcessExpression⟩ ⟶ ⟨FactorProcessExpression⟩ ⟨BinaryInfixProcessOperator⟩ ⟨FactorProcessExpression⟩

⟨ProcessExpression⟩ ⟶ ⟨FactorProcessExpression⟩

⟨Quantification⟩ ⟶ `forall` { ⟨VariableTyping⟩ , }+

⟨RationalConstant⟩ ⟶ ⟨DecimalDigit⟩+ . ⟨DecimalDigit⟩+

⟨RelationSpecialization⟩ ⟶ `partitioned by` { ⟨SubRelationName⟩ , }+ ;

⟨RelationshipBlock⟩ ⟶ `begin relationship class` ⟨UserDefinedRelationshipName⟩
⟨RelationshipGeneralization⟩*
⟨RelationshipSpecialization⟩*
⟨Components⟩
[⟨Attributes⟩ *]*
[⟨Predicates⟩ *]*
[⟨Events⟩ *]*
[⟨Process⟩ *]*
[⟨Axioms⟩ *]*
`end relationship class` ⟨UserDefinedRelationshipName⟩ ;

⟨RelationshipGeneralization⟩ ⟶`specialization of` ⟨SuperRelationshipName⟩ ;

⟨ReservedWord⟩ ⟶ `after` | `and` | `attributes` | `axioms` | `begin` | `bijection` | `by` | `class` | `components` | `creation` | `cycle` | `decompositions` | `deletion` | `div` | `empty` | `end` | `events` | `fail` | `fails` | `fixed` | `forall` | `function` | `functions` | `identifiers` | `if` | `In` | `initially` | `injection` | `inverse` | `keys` | `life` | `necessarily` | `not` | `object` | `of` | `only` | `or` | `partitioned` | `possibly` | `predicate` | `predicates` | `relationship` | `services` | `skip` | `specialization` | `specialized` | `succeeds` | `surjection` | `transactions` | `type` | `Value` | `;` | `,` | `::` | `(` | `)` | `[` | `]` | `->` | `<->` | `+` | `-` | `*` | `/` | `&` | `||` | `.` | `=` | `<` | `<=` | `>=` | `>` | `^` | `!=`

⟨ServiceBlock⟩ ⟶ `begin service` ⟨ServiceName⟩
[⟨Transactions⟩ *]*
[⟨Decompositions⟩ *]*
`end service` ⟨ServiceName⟩`;`

⟨SimpleCondition⟩ ⟶ { ⟨PredicateTerm⟩ `,` }+

⟨SimpleExpression⟩ ⟶ ⟨Constant⟩

⟨SimpleExpression⟩ ⟶ ⟨PrefixFunctionApplication⟩

⟨SimpleExpression⟩ ⟶ ⟨VariableName⟩

⟨SimpleExpression⟩ ⟶ `-` ⟨SimpleExpression⟩

⟨SimpleExpression⟩ ⟶ `(` ⟨Expression⟩ `)`

⟨SimpleSortExpression⟩ ⟶ ⟨SortName⟩ *[* ⟨ActualValueBlockParameterList⟩ *]*

⟨SortExpression⟩ ⟶ { ⟨SimpleSortExpression⟩ `*` }+

⟨SortName⟩ ⟶ ⟨PredefinedSortName⟩ | ⟨UserDefinedSortName⟩

⟨SpecialCharacterConstant⟩ ⟶ `\n` | `\r` | `\f` | `\t` | `\b` | `\"` | `\`⟨OctalDigits⟩

⟨StringConstant⟩ ⟶ `"` ⟨CharacterSpecification⟩* `"`

⟨SubRelationName⟩ ⟶ ⟨UserDefinedRelationshipClassName⟩

⟨SubSortExpression⟩ ⟶ ⟨SortExpression⟩

⟨SubjectExpression⟩ ⟶ ⟨Expression⟩

⟨SuperClassName⟩ ⟶ ⟨UserDefinedClassName⟩

⟨SuperRelationshipName⟩ ⟶⟨UserDefinedRelationshipName⟩

⟨SuperSortName⟩ ⟶ ⟨UserDefinedSortName⟩

⟨Transactions⟩ ⟶ `transactions` ⟨Transaction⟩*

⟨Transaction⟩ ⟶ ⟨TransactionName⟩ *[* ⟨EventFormalParameters⟩ *]* ;

⟨UniquenessConstraint⟩ ⟶ `identifiers` ⟨IdentifierConstraint⟩*

⟨UniquenessConstraint⟩ ⟶ `keys` ⟨KeyConstraint⟩*

⟨UsedDefinedSortName⟩ ⟶ ⟨Name⟩

⟨UserDefinedEventName⟩ ⟶ ⟨Name⟩

⟨UserDefinedFunctionName⟩ ⟶ ⟨Name⟩

⟨UserDefinedObjectClassName⟩ ⟶ ⟨Name⟩

⟨UserDefinedPredicateName⟩ ⟶ ⟨Name⟩

⟨UserDefinedRelationshipName⟩ ⟶ ⟨Name⟩

⟨UserDefinedSortName⟩ ⟶ ⟨Name⟩

⟨ValueBlockParameterList⟩ ⟶ [{ ⟨ValueBlockParameter⟩ , }+]

⟨ValueBlockParameter⟩ ⟶ `value type` ⟨UserDefinedSortName⟩ |
`function` ⟨UserDefinedFunctionName⟩ *[* ⟨FormalParameterList⟩ *]*
: ⟨SortExpression⟩ |
`predicate` ⟨UserDefinedPredicateName⟩ *[*⟨FormalParameterList⟩ *]*

⟨ValueBlock⟩ ⟶ `begin value type` ⟨UserDefinedSortName⟩ *[* ⟨ValueBlockParameterList⟩ *]*
⟨ValueGeneralization⟩*
⟨ValueSpecialization⟩*
[⟨Functions⟩ *]*
[⟨ValuePredicates⟩ *]*
[⟨Axioms⟩ *]*
`end value type` ⟨UserDefinedSortName⟩ ;

⟨ValueGeneralization⟩ ⟶ `specialization of` { ⟨SuperSortName⟩ , }+ ;

⟨ValuePredicateDeclaration⟩ ⟶ ⟨PrefixForm⟩ ;

⟨ValuePredicates⟩⟶ `predicates` ⟨ValuePredicateDeclaration⟩*

⟨ValueSpecialization⟩ ⟶ `specialized by` { ⟨SubSortExpression⟩ , }+ ;

⟨VariableName⟩ ⟶ ⟨Name⟩

⟨VariableTyping⟩ ⟶ { ⟨VariableName⟩ , }+ : ⟨SortExpression⟩

⟨WhiteSpace⟩ ⟶ ⟨WhiteSpaceElement⟩+

⟨ZeroConstant⟩ ⟶ ⟨Zero⟩+

⟨Zero⟩ ⟶ 0

Bibliography

[1] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[2] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press/Addison Wesley, 1989.

[3] A. Blom. CMW –Entity Relationship Editor: User Manual. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[4] R.B. Feenstra. Experimental GNU Emacs support for creating CMSL specifications. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, 1993.

[5] L.T.F. Gamut. *Logic, Language and Meaning 2: Intensional Logic and Logical Grammar.* University of Chicago Press, 1991. L.T.F. Gamut is a pseudonym for J.F.A.K. van Benthem, J. Groenendijk, D. de Jongh, M. Stokhof, and H. Verkuyl.

[6] J.A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park, CA 94025, U.S.A., 1988.

[7] D. Harel. Dynamic logic. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic II*, pages 497–604. Reidel, 1984.

[8] D. Kozen and J. Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 789–840. Elsevier Science Publishers, 1990.

[9] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[10] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In M. Nivat and H. Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 2: Rewriting Techniques*, pages 297–367. Academic Press, 1989.

[11] R.J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models.* PhD thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1990.

[12] R.J. Wieringa. A conceptual model specification language (CMSL Version 2). Technical Report IR-248, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, May 1991.

[13] R.J. Wieringa. A method for building and evaluating formal specifications of object-oriented conceptual models of database systems (`MCM`). Technical Report IR-340, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[14] R.J. Wieringa and R.B. Feenstra. The university library document circulation system specified in LCM 3.0. Technical Report IR-343, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[15] R.J. Wieringa and P.A. Spruit Jonge, W. de. Roles and dynamic subclasses: a modal logic approach. Technical Report IR-341, Faculty of Mathematics and Computer Science, Vrije Universiteit, December 1993.

[16] R.J. Wieringa and P.A. Spruit Jonge, W. de. Roles and dynamic subclasses: a logic and methodology. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, In preparation.

Index