Larch/Smalltalk: A Specification Language for Smalltalk

Yoonsik Cheon
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports
Part of the Programming Languages and Compilers Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/37

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Larch/Smalltalk: A Specification Language for Smalltalk

Abstract
Larch/Smalltalk is a Larch interface specification language for Smalltalk with subtype relations. As a Larch-style language it benefits from two-tiered approach to specifications; separation of concerns, division of effort, and reusability. Subtype relationships helps to reuse and modularize specifications. A unit of specification in Larch/Smalltalk is called a type, which describes an abstraction of a set of Smalltalk classes. Complex specifications can be constructed by defining a type to be a subtype of other types, called its supertypes, thereby, inheriting their specifications. Specifications can also be parameterized to specify a family of related types. To encourage specifications to be used in the programming process, specification development tools have been implemented in Smalltalk. They are integrated in the Smalltalk-80 system. Using these tools, a portion of Smalltalk system classes and a part of the tools themselves have been specified.

Disciplines
Programming Languages and Compilers
Larch/Smalltalk: A Specification
Language for Smalltalk

TR91-15
Yoonsik Cheon

June, 1991
Larch/Smalltalk:
A Specification Language for Smalltalk

Yoonsik Cheon

TR #91-15
June, 1991

Keywords: specification, Smalltalk, pre-condition, post-condition, subtype, polymorphism, type checking.


Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040, USA
Larch/Smalltalk: A Specification Language for Smalltalk

by

Yoonsik Cheon

A Thesis Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Major: Computer Science

Approved:

Members of the Committee:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa
1991
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** .................................................. viii

**ABSTRACT** ............................................................... ix

**CHAPTER 1. INTRODUCTION** .......................................... 1

1.1 Background ........................................................... 2

1.1.1 Larch-style two-tiered specifications .......................... 2

1.1.2 The Larch Shared Language ................................... 3

1.1.3 Smalltalk ........................................................ 6

1.2 Specifications in Larch/Smalltalk ................................. 8

1.2.1 An overview of Larch/Smalltalk ............................... 8

1.2.2 Supporting tools ............................................... 11

1.3 Related Work ...................................................... 12

**CHAPTER 2. LARCH/SMALLTALK** ................................. 16

2.1 Method Specifications .............................................. 17

2.1.1 Assertions .................................................... 20

2.1.2 Mutation ...................................................... 26

2.1.3 New objects .................................................. 27

2.1.4 Comments ..................................................... 29

2.2 Type Specifications ............................................... 29
2.2.1 Subtyping ........................................ 33
2.2.2 Mutation .......................................... 35
2.2.3 Invariants ......................................... 36
2.3 Parameterized Type Specifications ...................... 39
  2.3.1 Subtyping .......................................... 42
2.4 Block Specifications .................................. 44
  2.4.1 Subtyping .......................................... 47
2.5 Summary ............................................. 48

CHAPTER 3. SUPPORT TOOLS FOR LARCH/SMA... 50

3.1 System Type Browser ................................ 50
  3.1.1 Viewing existing specifications .................. 52
  3.1.2 Modifying and adding specifications ............... 57
  3.1.3 Specialized browsers .............................. 58
  3.1.4 Viewing the shared component ..................... 62
  3.1.5 Browsing implementing classes and direct subtypes (supertypes) 63
  3.1.6 Saving specifications ............................ 63
3.2 System Trait Browser ................................ 65

CHAPTER 4. CONCLUSIONS ............................. 67

BIBLIOGRAPHY .......................................... 70

APPENDIX A ........................................... 74

APPENDIX B ........................................... 78

APPENDIX C ........................................... 81
  C.1 System Type Browser ................................ 81
C.1.1 Type categories pane ........................................ 81
C.1.2 Type names pane ........................................... 83
C.1.3 Message categories pane ................................. 84
C.1.4 Message selectors pane ................................. 84
C.1.5 Implementing classes pane ............................... 85
C.1.6 Direct subtypes (supertypes) pane ....................... 86
C.2 System Trait Browser ........................................ 86
C.2.1 Trait categories pane ................................. 87
C.2.2 Trait names pane ........................................ 88
LIST OF TABLES

Table 2.1: Grammatical notations that extend BNF .............................................. 17
Table 2.2: Precedence of operators ................................................................. 24
Table 3.3: File name conventions ................................................................. 64
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Trait SetOfE</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>A definition of a method with selector size.</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>A Larch/Smalltalk specification for type IntegerSet</td>
<td>10</td>
</tr>
<tr>
<td>1.4</td>
<td>A system trait browser (left) and a system type browser (right)</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>A specification of type IntegerSet</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>Trait NatTrait</td>
<td>38</td>
</tr>
<tr>
<td>2.7</td>
<td>A parameterized type specification Set</td>
<td>40</td>
</tr>
<tr>
<td>2.8</td>
<td>Trait Block1 specifying blocks taking one argument.</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>A system type browser</td>
<td>51</td>
</tr>
<tr>
<td>3.10</td>
<td>A specification of type QualifiedIdentifierNode</td>
<td>54</td>
</tr>
<tr>
<td>3.11</td>
<td>Message list generated by using find method for type AssertionNode</td>
<td>56</td>
</tr>
<tr>
<td>3.12</td>
<td>A method specification template (left) and an error report (right)</td>
<td>58</td>
</tr>
<tr>
<td>3.13</td>
<td>A type category browser (left) and a type browser (right)</td>
<td>59</td>
</tr>
<tr>
<td>3.14</td>
<td>A message category browser (left) and a message browser (right)</td>
<td>60</td>
</tr>
<tr>
<td>3.15</td>
<td>A type hierarchy browser opened on type QualifiedIdentifierNode</td>
<td>61</td>
</tr>
<tr>
<td>3.16</td>
<td>Method specifications (bottom) and implementations (right)</td>
<td>62</td>
</tr>
</tbody>
</table>
Figure 3.17: A system trait browser .............................. 66
Figure 3.18: A trait category browser (left) and a trait browser (right) . 66
Figure A.19: Built-in trait \textit{Equality} ................................. 74
Figure A.20: Built-in trait \textit{Boolean} ................................. 75
Figure A.21: Trait \textit{State} ............................................. 76
Figure A.22: Trait \textit{Mapping} ........................................... 77
Figure C.23: Yellow button menus of a system type browser ............ 82
Figure C.24: Yellow button menus of a system trait browser ............ 86
ACKNOWLEDGEMENTS

I would like to express my foremost thanks to my advisor, Gary T. Leavens, for his sustaining guidance and encouragement throughout this work. I also wish to thank my parents with all my heart for their unceasing moral support and confidence in me.
ABSTRACT

Larch/Smalltalk is a Larch interface specification language for Smalltalk with subtype relations. As a Larch-style language it benefits from two-tiered approach to specifications; separation of concerns, division of effort, and reusability. Subtype relationships helps to reuse and modularize specifications.

A unit of specification is called a type, which describes an abstraction of a set of Smalltalk classes. Complex specifications can be constructed by defining a type to be a subtype of other types, called its supertypes, thereby, inheriting their specifications. Specifications can also be parameterized to specify a family of related types. To encourage specifications to be used in the programming process, specification development tools have been implemented in Smalltalk. They are integrated in the Smalltalk-80 system. Using these tools, a portion of Smalltalk system classes and a part of the tools themselves have been specified.
CHAPTER 1. INTRODUCTION

Object-oriented techniques are touted as a practical programming methodology that encourages software reuse and modular design. In Smalltalk, code reuse is achieved by defining one class to be a subclass of another class, thereby, inheriting its data definition and operations. Classes and subclass relationships are also used to structure Smalltalk programs.

To reuse code, one must understand its behavior precisely. However the sheer volume of existing classes, their complex interactions, and their implementation details frustrate programmers who, by inspecting code, try to understand potentially useful classes and their operations. Informal documents and program comments may be something of a relief. However, their imprecise and verbose nature prevent them from being much help. Worse, classes are usually grouped in such a way as to give a high degree of code sharing, not according to their conceptual relationships. This make it difficult for programmers to understand the behaviors of existing classes thereby causing poor code reuse.

In an effort to solve this problem, we have designed a Larch-style specification language for Smalltalk, called Larch/Smalltalk. This is a formal language with explicitly and precisely defined syntax. The semantics of Larch/Smalltalk could be given formally, although we will not do so here. Specifications written in Larch/Smalltalk
are thus more precise than informal specifications. Also, our specifications are modularized in a hierarchy based on their conceptual relationships.

Section 1.1 below gives the background of the work, including the Larch-style two-tiered approach to specification, and an overview of the Larch Shared Language and Smalltalk programming language. Section 1.2 gives an overview Larch/Smalltalk and its supporting tools, and Section 1.3 surveys related work.

1.1 Background

1.1.1 Larch-style two-tiered specifications

The two-tiered approach [37, 38] to program specification separates the specification of underlying abstractions from the specification of state transformations. Thus, a specification of each program module has a component on each tier. The state transformations, called the interface components, are written in a predicative language using pre- and post-conditions, and describe the effect of operation executions on program state (e.g., change of an object’s value or creation of a new object). The underlying abstractions, called the shared components, are written in the style of an equational algebraic specification, and describe intrinsic properties that are independent of the model of computation (e.g., a set is a collection of unordered, not-duplicated elements). The philosophy behind this approach is summarized in [38] as:

We believe that for specifications of program modules, the environment in which a module is embedded, and hence the nature of its observable behavior, is likely to depend in fundamental ways on the semantic primitives of the programming languages. ⋯ Thus we intentionally make an interface language dependent on a target programming language, and
keep the shared language independent of any programming language. To capitalize on our separation of a specification into two tiers, we isolated programming language dependent issues — such as side effects, error handling, and resource allocation — into the interface language component of a specification (pp. 3 in [38]).

In the Larch family of specification languages [22, 23], the shared components are specified in the Larch Shared Language (LSL) [19, 20, 21], and the interface components are specified in Larch Interface Languages such as Larch/CLU [37, 38], Larch/Pascal [22], Larch/Ada [15], and Larch/Avalon [40]. Larch/Smalltalk can be thought of as a member of the family of Larch Interface Languages whose target language is Smalltalk.

Main features of the Larch family of specification languages are (1) the separation of concerns and the division of effort between language independent and language dependent issues, (2) extensive checking of specifications using powerful theorem provers as specifications are being constructed [8, 9], (3) incremental construction of specifications from other specifications and reusability of shared components by different interface languages. In addition to above, Larch/Smalltalk specifications are constructed interactively and incrementally using supporting tools in the interactive graphical environment of the Smalltalk-80 system.

1.1.2 The Larch Shared Language

Since the shared components of our specifications are specified in the Larch Shared Language (LSL) [21], a brief overview of LSL is given in this section. The unit of specification in the LSL is called a trait. A trait may describe an abstract data type or may encapsulate a property shared by several data types. Trait SetOfE shown
SetOfE: trait
  includes Integer
  introduces
    empty: \rightarrow S
    insert: S, E \rightarrow S
    delete: S, E \rightarrow S
    include: S, E \rightarrow \text{Bool}
    isEmpty: S \rightarrow \text{Bool}
    size: S \rightarrow \text{Int}
  asserts
    S \text{ generated by} \ empty, \ insert
    S \text{ partitioned by} \ isEmpty, \ include
    \forall s: S, e, e1: E
    \hspace{1em} \text{delete}(empty,e) == empty
    \hspace{1em} \text{delete}(\text{insert}(s,e),e1) ==
    \hspace{2em} \text{if} \ e = e1 \text{ then} \text{delete}(s,e1) \text{ else} \text{insert}(\text{delete}(s,e1),e)
    \hspace{1em} \text{not}(\text{include}(\text{empty},e))
    \hspace{1em} \text{include}(\text{insert}(s,e),e1) ==
    \hspace{2em} \text{if} \ e = e1 \text{ then} \text{true} \text{ else} \text{include}(s,e1)
    \hspace{1em} \text{size}(\text{empty}) == 0
    \hspace{1em} \text{size}(\text{insert}(s,e)) ==
    \hspace{2em} \text{if} \ \text{include}(s,e) \text{ then} \text{size}(s) \text{ else} \text{size}(e) + 1
    \hspace{1em} \text{isEmpty}(s) == \text{size}(s) == 0

Figure 1.1: Trait SetOfE

in Fig. 1.1, describes mathematical sets, and is similar to a conventional algebraic specification.

A trait consists of two parts; operators declarations and assertions. A set of operators are declared following the keyword introduces. An operator is given its name and signature (the sorts of its domains and range). These signatures are used to sort-check terms of the equational axioms in the assertion part. The assertion part which is preceded by the keyword asserts, specifies a set of constraints to the
operators by means of equations and other clauses. An equation consists of two
terms of the same sort, separated by \(==\). Equations of the form \(\text{term} == \text{true}\) can
be abbreviated by simply writing the term; thus the third equation in the trait \text{SetOfE}
is an abbreviation for \(\text{not} (\text{include} (\text{empty}, e)) == \text{true}\).

A trait denotes a \textit{theory}\(^1\) in typed first-order logic with equality. Each theory
contains the trait’s assertions, the conventional axioms of first-order logic, everything
that follows from them, and nothing else. This means that the formulas in the theory
follow only from the presence of assertions in the trait — never from their absence. A
theory is strengthened by \textbf{generated by} and \textbf{partitioned by} clauses. A \textbf{generated
by} clause adds an inductive inference rule to a trait’s theory. For example, saying
that sort \(S\) is \textbf{generated by} a set of operators, \(\text{Ops}\), asserts that each term of sort
\(S\) is equal to a term whose outmost operator is in \(\text{Ops}\). In the \text{SetOfE} example,
all values of sets can be denoted by terms using only the operators, \texttt{empty} and
\texttt{insert}. A \textbf{partitioned by} clause asserts that all distinct values of a sort can be
distinguished by a given list of operators. Terms that are not distinguished using any
of the partitioning operators of their sorts are equal. In \text{SetofE} trait, for example,
\(\text{insert} (\text{insert} (\text{empty}, 0), 1)\) and \(\text{insert} (\text{insert} (\text{empty}, 1), 0)\) denote the same value, i.e.,
the set (in mathematical sense) with two elements \(0\) and \(1\).

The ISL also provides ways of combining traits. One way of combining traits
is by using the \textbf{includes} clause. A trait that includes another trait is textually
expanded to contain all operator declarations, \textbf{generated by} clauses, \textbf{partitioned
by} clauses, and axioms of the included trait. The meaning of the including trait is
the meaning of the textually expanded trait. In the trait \text{SetOfE}, the signature and

\(^1\)A \textit{theory} is a set of formulas without free variables.
the meaning of + comes from the included trait `Integer`. Boolean operators (true, false, not, \&, |, =>, <=>) and some heavily overloaded operators (if-then-else, =, \sim=) are built into the language; that is to say, traits defining these operators are implicitly included in every trait.

### 1.1.3 Smalltalk

Our specification language is targeted to Smalltalk, although most constructs can be used without modification by other object-oriented languages with inheritance. Smalltalk is an untyped, pure object-oriented programming language in that everything, from integers to classes to the execution state is an object. Since everything is an object, the only operation needed is message sending. Below we summarize unique features of Smalltalk that must be captured by our specification language. A full description of Smalltalk can be found in [12, 13, 29, 30].

A Smalltalk program consists of a set of `classes`. A class is a module that implements a data abstraction in Smalltalk. A class describes a set of objects\(^2\), called `instances` of the class, by defining a set of methods, which implement the `messages` to which instances of that class respond. A message can be unary, binary, or keyword. A unary message takes no argument, a binary takes one argument, and a keyword takes one or more arguments. Fig. 1.2 shows a unary method with selector `size` that may be defined in class `Set`. The `pseudo-variable self` refers to the receiver of the message that invokes the method. For example, if the method is invoked by the expression, `s size` (where `s` is an instance of the class `Set`), `self` refers to `s`. The up-

\(^2\)Smalltalk defines two kinds of objects; `mutable` and `immutable`. An object is mutable if it has time varying states, otherwise it is immutable.
size

“Answer how many elements the receiver contains.”

<table>
<thead>
<tr>
<th>tally</th>
</tr>
</thead>
</table>
tally ← 0.
self do: [each | tally ← tally + 1]
↑ tally

Figure 1.2: A definition of a method with selector size.

arrow (↑) preceding the last expression says that the value of this expression will be returned as the result of the method evaluation. If omitted, the receiver is returned by default.

Smalltalk’s control structures are all implemented in terms of message sending and blocks\(^3\). For example, the ifTrue:ifFalse: message, Smalltalk’s equivalent to if-then-else statement, is sent to a Boolean object. If the receiver is true, i.e., if the receiver is an instance of the class True, the “true block” is evaluated; otherwise (the receiver is false, i.e., it’s an instance of the class False) the “false block” is evaluated. Iterations are implemented in a similar way using blocks and recursion.

In Smalltalk, a class is defined in terms of its difference from another class, called its superclass. All classes that are so defined with respect to a particular class are called its subclasses. A subclass inherits all the methods of its superclasses; it may override inherited methods or add new methods. For example, the class Set is a subclass of the class Collection, which is a subclass of the class Object, the ultimate superclass of all classes in Smalltalk. So the class Set inherits (unless it overrides) all

\(^3\)A block is a closure, i.e., it contains a parameterized code and an environment to look up variables that are referred in the code, but are not parameters.
the methods defined in the class `Collections` and those methods inherited by `Collection` from the class `Object`.

Since a class is represented at runtime by an object, there is a different class that describes each such `class object`. A class that describes class objects is called a `metaclass`. A metaclass usually defines how to create instances of its instances. For example, `Set class`, the metaclass for the class `Set`, defines two class methods `new` and `new:`, which return new instances of the class `Set` with default size (defined by the system) and with size given by an input argument respectively.

### 1.2 Specifications in Larch/Smalltalk

Following the Larch-style two-tiered approach, a Smalltalk program module is specified in two levels; in the shared level, underlying abstractions are specified algebraically, and in the interface level, the behavior of the module is axiomatically specified using the terms provided by the shared component. Though different specification languages could be used in the shared level, we use the Larch Shared Language [21], which was summarized in Section 1.1.2. Our specification language, Larch/Smalltalk, is used to specify the behavior of Smalltalk program modules in the interface level, and an overview is given in Section 1.2.1 below. Section 1.2.2 describes supporting tools for Larch/Smalltalk.

#### 1.2.1 An overview of Larch/Smalltalk

Larch/Smalltalk is a two-tiered specification language belonging to the family of Larch interface languages [22, 23, 37, 38, 40]. The basic unit of specifications in Larch/Smalltalk is an `abstract data type`, henceforth, called a `type` for short. A type
is an abstraction of a set of related Smalltalk classes, characterized by a behavioral specification. Figure 1.3 shows a specification of type \texttt{IntegerSet}. \texttt{IntegerSet} is specified to be a mutable type (i.e., its objects may mutate their values), and is a subtype of the type \texttt{ObjectWithEqual}. It defines six \textit{instance methods}\footnote{\textit{Instance methods} are those methods that define the messages that are sent to the objects of type \texttt{IntegerSet}; \textit{meta methods} are those methods that define the messages that are sent to the object that describes the type \texttt{IntegerSet} itself (class objects in Smalltalk). Instance methods correspond to instance methods in Smalltalk; meta methods correspond to \textit{class methods} in Smalltalk.} (\texttt{insert:}, \texttt{delete:}, \texttt{isIn:}, \texttt{size}, \texttt{isEmpty}, and \texttt{=} ) and one \textit{meta method} (\texttt{new}). The link to the shared component is given by the \texttt{usedTrait} clause, which specifies the \textit{used trait} and the \textit{type-to-sort mapping}. The used trait \texttt{SetOfE} (see Figure 1.1) provides sort names and operator names to be used in method specifications, and the type-to-sort mapping determines the values over which objects of \texttt{IntegerSet} can range (i.e., the abstract values of type \texttt{IntegerSet}). All \texttt{IntegerSet} objects are restricted to values denotable by terms of sort \texttt{S} which is defined in the trait \texttt{SetOfE}.

A method specification consists of a header and a body. The header gives information necessary to send the defined message while the body describes the behavior of the method. The header which is similar to that of Smalltalk methods in form, lists input and return arguments and their types. The body contains two predicates (\texttt{requires} and \texttt{ensures} clauses) that correspond to a pre-condition on the \textit{initial state}, the state when the method is invoked and, a post-condition on the \textit{final state}, the state when the evaluation of the method terminates. Terms in pre- and post-conditions are constructed from operators provided by the used trait. An optional \texttt{modifies at most} clause list those objects that may mutate as the result of the method evaluation.
type IntegerSet
    supertypes ObjectWithEqual
    mutation true
    usedTrait SetOfE(IntegerSet for S, Integer for E)

instance methods
    insert: e :< Integer
        modifies at most self
        ensures self.post = insert(self.pre,e)

    delete: e :< Integer
        requires include(self.pre,e)
        modifies at most self
        ensures self.post = delete(self.pre,e)

    isIn: e :< Integer
        returns b :< Boolean
        ensures b = include(self.pre,e)

    size
        returns n :< Integer
        ensures n = size(self.pre)

    isEmpty
        returns b :< Boolean
        ensures b = isEmpty(self.pre)

    = s :< IntegerSet
        returns b :< Boolean
        ensures b = forall(i: Int)[include(self.pre,i) <=> include(s,i)]

meta methods
    new
        returns s :< IntegerSet
        ensures s = empty & new(s)

Figure 1.3: A Larch/Smalltalk specification for type IntegerSet
For example, see the `delete` method specified in the type `IntegerSet`. The `self` that appears in the method body, denotes the receiver, i.e., an object of `IntegerSet` to which the message is sent. The `self.pre` denotes the value of `self` in the initial state; `self.post` denotes the value of `self` in the final state. The pre-condition of `delete` is satisfied if the input argument `e` is an element of the receiver, `self`. The post-condition of `delete` contains an assertion about the final value of the receiver in terms of its initial value and the value of the argument `e`. It asserts that the receiver will not contain as its element the input argument `e`. That is to say, if `e` is one of the elements of the receiver, it will be removed from the receiver. The modifies at most clause asserts that this method may mutate only the receiver, nothing else. The operators names, `delete`, `=`, and `&`, all come from the used trait `SetOfE` (see Fig. 1.1).

In Larch/Smalltalk, one can also specify a parameterized type, a type specification with `type parameters`. A parameterized type specification specifies an infinite number of types. For example, instead of separately specifying types `IntegerSet`, `CharacterSet`, `StringSet`, etc., a parameterized version of type `Set` can be specified once and for all. Then, this specification can be instantiated by a type denoting the element type, thereby, creating new specifications such as `Set(Integer)`, `Set(Character)`, `Set(String)`, and so on. Parameterized type specifications are extended to specify block types (see Section 2.4).

### 1.2.2 Supporting tools

Specifications in Larch/Smalltalk are expected to be developed in an interactive graphical environment with an editor, syntax- and sort-checker, specifications manager, printing utilities, and a window system. System type browsers and system
trait browsers provide this environment (see Fig. 1.4). They were implemented in Smalltalk and integrated in the Smalltalk-80 programming environment. A system type browser allows one to view, modify, enter, delete, and check specifications in Larch/Smalltalk; a system trait browser does similar work on the shared components of specifications, i.e., traits specified in the Larch Shared Language. Major features of these browsers are:

- To enter, view, modify, delete specifications.
- To syntax- and sort-check specifications.
- To save and retrieve specifications to and from external files.
- To browse implementations (Smalltalk classes and their methods) of specifications both in the type level and in the method level.
- To browse the corresponding shared components of a type directly from interface specifications.

Using these tools, specifications are expected to be developed interactively and incrementally. System type browsers and system trait browsers will be explained in Chapter 3.

1.3 Related Work

In the past, formal specifications have been used to describe simple programs and abstract data types, leading to two different approaches, referred to as “operational” and “definitional.” A survey of and introduction to these approaches can
be found in [33], [34] and [39]. In the operational approach, one gives a method of constructing the program or abstract data type. Some examples of the operational approach are Parnas's work on state machines [35] and Jones's model-oriented specifications [28]. In the definitional approach of specifying a program or an abstract data type, one gives a list of its desired properties, not a method for constructing it. The definitional approach can be broken into two categories, referred to as "axiomatic" and "algebraic." In the axiomatic approach, two predicates, called pre- and post-conditions, are used for the specification of the input-output behavior of programs and each operation of an abstract data type [27]. The algebraic approach uses axioms to specify properties of programs and abstract data types, but the axioms are restricted to equations [10, 11, 16, 17, 18, 24]. This approach defines data types to be heterogeneous algebras [2].

Larch family of specification languages are related to both the operational and
definitional approaches. The Larch Shared Language (LSL) [21] is based on the algebraic approach, while the interface languages, Larch/CLU [37, 38], Larch/Pascal [22], Larch/Ada [15], Larch/Avalon [40], and our specification language Larch/Smalltalk, are similar to the model-oriented specifications. One important difference between Larch/Smalltalk and other interface languages is that Larch/Smalltalk has the notion of subtypes, thereby, allowing reuse of specifications in the interface level and modularization of specifications according to their conceptual relationships.

Recently there has been much effort on object-oriented specification languages and on ways of specifying and verifying programs in object-oriented programming languages. GSBL [7] is similar to conventional algebraic specification languages with the notion of classes and inheritance. Object-Z [6] is an extended version of the specification language Z [26], which uses the class concept to encapsulate the description of an object's state with its related operations. Complex specifications are then constructed using class inheritance and instantiation. One major difference between these two specification languages and Larch/Smalltalk is that while they are based on the notion of classes and subclasses, ours are based on types and subtypes [25, 36]. Also, specifications written in GSBL and Object-Z have no simple ways to specify side-effects and error-handlings. Bear et al. [1] designed a graphical notation called ObjectCharts, which combines object-oriented analysis and design techniques and state charts to give a diagrammatic specification for object-oriented systems. ObjectCharts can be well suited to describe general structure and behavior of object systems. However, it has neither subtype nor subclass mechanisms at all. Leavens [32, 31] proposed a modular way of specifying and verifying object-oriented programs using subtyping relationships. He argues that if subtype relationships satisfy cer-
tain semantic constraints (referred to as *simulation relationships*), a new type can be added to a program without respecifying or verifying unchanged modules — if the new type is a subtype of existing ones. His approach can be adapted to verify and reason about Smalltalk programs specified in Larch/Smalltalk if the simulation relationships are preserved among subtype relations, which is specifiers’ responsibility. Larch/Smalltalk requires only syntactic subtyping rules which is weaker than semantic ones (see Section 2.2.1).
CHAPTER 2. LARCH/SMAFULLTALK

In this chapter, we present the syntax and informal semantics of Larch/Smalltalk, and some examples. We use extended BNF with conventions shown in Table 2.1 to define the syntax. Note that nonterminals (meta symbols) are italicized, keywords are bold-faced, and terminals are printed in a roman font. A vertical bar (|) and square brackets ([, ]) are used for both as terminals and as nonterminals; those for terminals are underlined, i.e., \|, \[, and \] are a terminal vertical bar and terminal brackets (see Section 2.1.1).

A unit of specification in Larch/Smalltalk is a type, which is implemented by a set of Smalltalk classes. A type specification consists of a set of method specifications, which describe the messages to which objects of that type can respond. These method specifications correspond to method definitions of the implementing classes. Specifications of methods are described in Section 2.1. In Section 2.2, type specifications are explained; they are extended to parameterized type specifications in Section 2.3. In Section 2.4, more complex types, block types, are specified as a variation of parameterized types. Finally, Section 2.5 gives a summary of this chapter. The details of the Larch/Smalltalk syntax are given in Appendix B.
Table 2.1: Grammatical notations that extend BNF

<table>
<thead>
<tr>
<th>notation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{e}</td>
<td>grouping</td>
</tr>
<tr>
<td>[e]</td>
<td>optional e</td>
</tr>
<tr>
<td>e*</td>
<td>zero or more e’s</td>
</tr>
<tr>
<td>e*,</td>
<td>zero or more e’s, separated by commas</td>
</tr>
<tr>
<td>e+</td>
<td>one or more e’s</td>
</tr>
<tr>
<td>e+,</td>
<td>one or more e’s, separated by commas</td>
</tr>
<tr>
<td>alpha</td>
<td>the nonterminal symbol alpha (e.g., formalId)</td>
</tr>
<tr>
<td>alpha</td>
<td>the keyword alpha (e.g., returns)</td>
</tr>
<tr>
<td>alpha</td>
<td>the terminal alpha (e.g., self)</td>
</tr>
</tbody>
</table>

2.1 Method Specifications

Method specifications are part of type specifications (see Section 2.2), and describe the messages that can be sent to the objects of the specified type. The behavior of a method is specified by describing the relationship between the inputs and the output (return value) by giving a pair of constraints [27]. Provided that the actual message arguments satisfy the input constraints, the return value is guaranteed to satisfy the output constraints. The input constraints are called the pre-condition; the output constraints are called the post-condition. The example below, shows a unary method specification with selector choose, that can be specified in type IntegerSet (see Fig. 1.3 on page 10).

```plaintext
choose
returns  i <: Integer
requires not(isEmpty(self))
ensures   include(self,i)
```
The `self` in both the pre- and post-conditions denotes the receiver, i.e., the object to which the message `choose` is sent. The method `choose` takes no input argument and returns an object of type `Integer` only when the receiver is not empty. The returned object is one of the elements of the receiver. Operator names like `isEmpty` and include, come from the trait `SetOfE` (see Fig. 1.1 on page 4), the shared component of specification `IntegerSet`.

Formally, a method specification consists of two parts:

1. a header giving the name of the specified method, the input arguments and their types, and the output (return) argument and its type.

2. a body that specifies the pre- and post-conditions.

The header provides the information necessary to send the specified message while the body describes the behavior of the method, i.e., the effect of the corresponding message sending. The header is similar to that of Smalltalk except that we decorate the input formals with their types and specify explicitly the return formal and its type. The return formal and its type is optional; if omitted, the receiver is assumed to be returned. As in Smalltalk, a method specification can be unary, binary, or keyword. The body consists of a pair of assertions in the first-order predicate calculus; `requires` clause and `ensures` clause. A `requires` clause specifies the pre-condition that must hold to invoke the specified method, or equivalently to send the specified message. An omitted `requires` clause is interpreted as equivalent to “`requires` `true`”. An `ensures` clause states the post-condition that the specified method must establish upon termination, i.e., the post-condition is guaranteed to hold when the method evaluation is completed. Semantically, a method specification says that if
the specified method is invoked in a state in which the pre-condition holds, the
method invocation (evaluation) terminates and the post-condition is satisfied in that
termination state.

Syntax

The nonterminal assertion will be described in the following section, Section 2.1.1.
See Appendix B for the nonterminal identifier.

\[
\begin{align*}
\text{methodSpec} & ::= \text{msgPattern} \ [\text{returns} \ \text{formalDecl}] \ \text{methodBody} \\
\text{msgPattern} & ::= \text{unary} \mid \text{binary} \mid \text{keyword} \\
\text{methodBody} & ::= \ [\text{preCond}] \ \text{postCond} \\
\text{unary} & ::= \text{msgId} \\
\text{binary} & ::= \text{binarySelector} \ \text{formalDecl} \\
\text{keyword} & ::= \{ \text{msgId} : \text{formalDecl} \}^+ \\
\text{formalDecl} & ::= \text{formalId} :< \text{type} \\
\text{type} & ::= \text{typeId} \\
\text{binarySelector} & ::= \text{opChar} \ [\text{opChar}] \\
\text{preCond} & ::= \text{requires} \ \text{assertion} \\
\text{postCond} & ::= \text{ensures} \ \text{assertion} \\
\text{msgId} & ::= \text{identifier} \\
\text{typeId} & ::= \text{identifier} \\
\text{formalId} & ::= \text{identifier}
\end{align*}
\]

Checking

For a method specification to be syntactically well-formed, it must satisfy fol-
lowing conditions:
1. The input formals and the return formal are distinct.

2. The return formal appears only in the post-condition.

3. Formals (including self\(^5\)) of the form \(f.post\) appear only in the post-condition.

4. The pre- and post-condition are well-formed.

See Section 2.1.1 for the explanation of \(post\), and well-formedness of the pre- and post-conditions.

2.1.1 Assertions

In this section we describe the constructs that we use to make assertions about the objects and their values in a state. These assertions appear in the pre- and post-conditions of method specifications and in the invariant clause (see Section 2.2.3) of type specifications.

An assertion is a predicate stated by first-order predicate calculus. Boolean connectives (not, \&, |, =>, and \(<=>\)), the universal quantifier (forall), and the existential quantifier (exist) are used to compose a predicate. In addition to these, several other identifiers and symbols may appear in an assertion:

1. an implicit input formal self that denotes the receiver to which the specified message is sent.

2. explicit input and return formal variables.

\(^5\)Self is an implicit input formal denoting the receiver.
3. locally bounded logical variables; e.g., \( n \) in `forall(n: Nat)[n \geq zero]`. The formals and the logical variables must be distinct and a logical variable is bound to the nearest declaration in nested quantified assertions.

4. sort identifiers (e.g., `Nat` in the above assertion (item 3)), operator identifiers (zero), and operator symbols (`\geq`) defined in the shared component (i.e., traits) (see Section 1.2 and Section 2.2) of the type specification in which the assertion appears.

All the above can be used in assertions of method specifications (i.e., pre- and post-conditions), however, only 3 and 4 are allowed in the `invariant` clause (see Section 2.2.3) of type specifications. In the method specifications that mutate their arguments (see Section 2.1.2), it is sometimes necessary to refer to the value of an object in two different states; the states before and after the method invocation. And it is also necessary to refer to the identity of an object, that is to say, the object itself, not its value. The state just before the invocation of a method is called the `initial state`; the state just after the completion of a method evaluation is called the `final state`. The value of an object in the initial state is called its `initial value`; the value in the final state is called its `final value`. Input formals (including `self`) and return formal are qualified with a value qualifier (`.pre` and `post`) to denote their values in a particular state; they can also be qualified with an object qualifier (`.obj`) to denote their object identities. The meanings of these qualifications are:

1. `.pre` denotes the initial value of an object.

2. `.post` denotes the final value of an object.
3. .obj denotes the object itself. For example, self.obj denotes the receiver itself, not its value.

Qualifications are often redundant, so we adopt certain default depending on the context in which an identifier appears. An unqualified input formal (including self) is, by default, qualified with .pre; an unqualified return formal is, by default, qualified with .post. Both in the modifies clause (see Section 2.1.2) and in the new clause (see Section 2.1.3), one always refers to objects, hence, identifiers in these clauses have .obj as their default qualifiers.

Syntax

See Appendix B for the nonterminals identifier and opChar.

```
assertion ::= true | false | not(assertion) | (assertion)
            | assertion connectives assertion
            | quantifier (varDecList) [assertion]
            | term = term | term ~ = term

term ::= varId | selfOrFormalId | (term)
       | opId[(term^+,:)] | term infixOp term
       | selfOrFormalId.pre | selfOrFormalId.post
       | selfOrFormalId.obj

connectives ::= & | | => | <=
quantifier ::= forall | exist

varDecList ::= { varId^+, ; sortId}^+,
selfOrFormalId ::= self | formalId
infixOp ::= opChar [opChar]
formalId ::= identifier
opId ::= identifier
```
sortId ::= identifier
varId ::= identifier

Larch/Smalltalk has a simple precedence scheme for operators used in assertions. While some of these operators are built into the language, most of them come from the shared component of the specification, i.e., user-defined traits specified in the LSL. The first are called built-in operators and the second are called user-defined operators. Built-in operators include the Boolean operators (true, false, not, &, |, =>, <=>) and two heavily overloaded equational operators (=, =~) (see traits Boolean and Equality in Appendix A). Table 2.2 shows the precedence among these operators. Note that:

- Built-in and user-defined prefix operators\(^6\) (including quantifiers) have the highest precedence.

- Prefix operators have higher precedence than infix operators.

- User-defined infix operators have higher precedence than built-in infix operators.

- Among built-in infix operators, equational operators (=, =~) have highest precedence, then, Boolean operators & and |, and finally Boolean operators => and <=>.

- Unparenthesized infix assertions with multiple operators of the same precedence, are associated from left to right. Thus, \(w \& x \mid y \& z\) is equivalent to

\(^6\)A zero argument operator (e.g., true, false) is thought as a prefix operator with null argument
Table 2.2: Precedence of operators

<table>
<thead>
<tr>
<th>built-in prefix operators (true, false, not)</th>
<th>built-in quantifiers (forall, exist)</th>
<th>user-defined prefix operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-defined infix operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>built-in infix operators</td>
<td>equal ( = )</td>
<td>not equal ( \neq )</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>( \mid )</td>
</tr>
<tr>
<td></td>
<td>=&gt;</td>
<td>( \langle \Rightarrow \rangle )</td>
</tr>
</tbody>
</table>

\(((w \& x) | y) \& z\).

- Parentheses can be freely used to override the above precedence rules.

Checking

The well-formedness of an assertion is defined in terms of its subterms. The subterms of an assertion (or a term), \( \alpha \), is inductively defined as:

1. \( \alpha \) is a subterm of itself.

2. If \( \alpha \) is of the form \( f(t_1, t_2, \ldots, t_n) \), where \( f \) is a built-in or user-defined prefix operator, and \( t_1, t_2, \ldots, t_n \) are of assertions (or terms), then \( t_1, t_2, \ldots, t_n \) are subterms of \( \alpha \).

3. If \( \alpha \) is the form \( t_1 \ f \ t_2 \), where \( f \) is a built-in or user-defined infix operator, and \( t_1 \) and \( t_2 \) are assertions (or terms), then both \( t_1 \) and \( t_1 \) are subterms of \( \alpha \).

4. If \( \alpha \) is a quantified assertion (forall, exist), then its assertion part is a subterm of \( \alpha \).
Each subterm denotes a value of a particular sort (defined in a trait). The sort of the value is called the nominal sort of the subterm. For example, terms true and false denote two Boolean values and their nominal sorts are sort Bool. The nominal sort of self is given by the type-to-sort mapping in the usedTrait clause (see Section 2.2) of the type specification in which it appears. The (nominal) sort of other formals are given in the same way in the specification of its type, or remapped in the type-to-sort mapping. The sort of “.obj” qualified identifier is that of the plain identifier concatenated with “.obj”. The sort for quantified subterms (forall, exist) are sort Bool. The sort for other subterms is the range sort of its out-most operator.

An assertion is said to be well-formed (or sort-checked in [37]) if for each of its subterms:

1. If the subterm is a logical variable or formal (including self), it is trivially sort-checked.

2. If the subterm is a zero argument operator (e.g., true, false), then the operator with null domain must be defined in the shared component.

3. If the subterm is of the form $f(t_1, t_2, \ldots, t_n)$, where $t_1, t_2, \ldots, t_n$ are assertions (or terms), then the shared component defines a prefix operator named $f$ whose domain is the same as the list of the sorts of the terms, $t_1, t_2, \ldots, t_n$ in that order.

4. If the subterm is of the form $t_1 f t_2$, where $t_1$ and $t_2$ are assertions (or terms), then the shared component defines an infix operator named $f$ whose domain is the same as the list of the sort of $t_1$ followed by that of $t_2$. 

5. If the subterm is a quantified assertions (forall, exist), then the sort of its assertion part is $Bool$.

To sort-check assertions that use built-in operators, the built-in traits `Boolean` and `Equality` (see Appendix A) are incorporated into the shared component of every type specification.

### 2.1.2 Mutation

In Smalltalk, methods can mutate objects, i.e., change their values. To specify this effect, we extend the syntax of method specifications to include a `modifies` clause in the body. A `modifies` clause asserts that only those objects that are listed in the clause may be mutated as the result of the evaluation of the specified method. This is a strong indirect assertion that no other objects except for those listed in the clause, are allowed to be mutated. This assertion is implicitly conjoined to the post-condition in the `ensures` clause. An omitted `modifies` clause is equivalent to the assertion “`modifies nothing`”, meaning no object are allowed to be mutated.

**Example**

```plaintext
intersect: s := IntegerSet
    modifies at most self
    ensures forall (i: Int) [include(self.post,i) <=> (include(self.pre,i) & include(s.pre,i))]
```

This method specification of type `IntegerSet` specifies that `intersect:` takes an argument of type `IntegerSet`, may mutate the receiver to make its value, in the final state, equal to the intersection of the value of the receiver in the initial state and that
of the argument, and return this modified receiver. The method \texttt{intersect:} may change the value of the receiver, but cannot mutate the input argument, \texttt{s}, nor any other objects.

\section*{Syntax}

See Section 2.1.1 for the nonterminals \texttt{preCond} and \texttt{postCond}. See Appendix B for the nonterminal \texttt{identifier}.

\begin{verbatim}
methodBody ::= [preCond] [modiList] postCond
modiList ::= modifies nothing
            | modifies at most selfOrFormalId+

selfOrFormalId ::= self | formalId
formalId ::= identifier
\end{verbatim}

\section*{Checking}

1. Only \texttt{self} and input formals can appear in the \texttt{modifies} clause and their types must be \texttt{mutable}. See Section 2.2.2 for an explanation of the mutability of types.

2. Identifiers in the \texttt{modifies} clause should not be value-qualified (i.e., with \texttt{.pre} or \texttt{.post}).

\subsection*{2.1.3 New objects}

In Smalltalk, methods can create new objects. This is specified by a \texttt{new} clause in the post-condition, which asserts that the objects listed in the clause are newly
created as the result of the invocation of the specified method. That is to say, these objects do not exist in the initial state, but do in the final state. If there is no `new` clause in the post-condition, the method will not create any new objects\(^7\).

Example

```plaintext
singleton: e :< Integer
  returns  s :< IntegerSet
    “Answer a new set with one element e.”
  ensures  s = insert(empty,e) & new(s)
```

This meta method specification of the type `IntegerSet` asserts that when the specified message `singleton:` is sent to `IntegerSet`, a new `IntegerSet` object (denoted by `s` in the specification) will be created and returned. The value of this newly created object is a set with one element `e`, the input argument.

Syntax

See Appendix B for the nonterminal `identifier`. See also Section 2.1.1 for the details of assertions.

```plaintext
assertion ::= ... | new(formalId+,
formalId ::= identifier
```

\(^7\)In addition to those listed in the `new` clause, the specified method may create other new objects in the intermediate states, but they are not visible in the final state. Technicali, they are temporary objects that exist only for the duration of the method evaluation. Formally, the set of objects in the final state is the union of the set of objects in the initial state and all those objects listed in `new` clauses.
Checking

1. A new clause can appear only in the post-condition.

2. Only input and return formals can be listed in the new clause. And they must not be value-qualified.

2.1.4 Comments

Comments can be given in method specifications by simply placing them inside a pair of double quotes (see the singleton: method specification in Section 2.1.3). They can be placed anywhere in the specifications where blanks are allowed.

2.2 Type Specifications

In this section, the syntax for type specifications is described with examples. The basic constructs are explained first to specify simple types, and they are extended to define more complex types.

A type is an abstraction of several Smalltalk classes, characterized by a behavioral specification. The term type here is similar to those found in [3, 14] and interfaces in [4]. A type specification can be implemented by more than one Smalltalk classes. For example, type Boolean is implemented by three Smalltalk classes, Boolean, True, and False. Specifying programs in terms of types, not in Smalltalk classes, allows one to structure specifications in a hierarchy based on their conceptual relationships, thus, leading to more understandable specifications; Smalltalk classes are usually organized in such a way to give a high degree of code sharing, not according to their logical relationships. A type specification can be thought of as a family of related
method specifications with a common used trait. The used trait is the shared component of the specification, and provides not only the abstract values of the specified type but also other symbols to be used in its method specifications. The method specifications of a type specification define the set of messages to which objects of the specified type can respond.

Fig. 2.5 shows a specification for type \texttt{IntegerSet}\textsuperscript{8}. The type \texttt{IntegerSet} is defined to be a direct subtype of type \texttt{ObjectWithEqual}, thereby, inheriting all instance method specifications of \texttt{ObjectWithEqual} (see Section 2.2.1). The used Trait clause gives the name of the used trait and a type-to-sort mapping. The used trait \texttt{SetOfE} (see Fig. 1.1 on page 4) provides both the abstract values of \texttt{IntegerSet} and abstract operator names (e.g., include, empty) to be used in specifying methods of \texttt{IntegerSet}.

\textsuperscript{8}A type name is used to denote for both the specified type and its specification.
The type-to-sort mapping, given by the `usedTrait` clause, is a mapping from type names to sort names. Type `IntegerSet` is mapped to sort S, and `Integer` to sort E. Due to this mapping, we know that type `IntegerSet` has such abstract values as empty, `insert(\text{empty}, 0)`, `insert(\text{empty}, 1)`, `insert(insert(\text{empty}, 0), 1)` and so on — terms of sort S.

Formally, a type specification has three parts;

1. a header giving the name of the specified type, the names of its direct supertypes
2. a link specifying the used trait and the mapping from types to sorts
3. a specification of each method.

Two kinds of methods are specified; *instance methods* and *meta methods*. An instance method defines a message that is sent to an instance of a type, i.e., an object of the type. A meta method corresponds to a Smalltalk class method, and defines a class message that is sent to a Smalltalk class object, not to an instance object. A method specification that specifies an instance methods is called an *instance method specification*; a specification that describes a meta method is called a *meta method specification*. A meta method specification usually specifies how to create an instance (object) of the specified type. A method specification is classified as an instance or a meta when it is specified using the supporting tools described in Chapter 3. There is no special syntactic construct that tell whether a method specification is an instance or a meta. In the previous example, method `isInt` is entered as an instance method specification, and `new` as a meta.
Syntax

See Section 2.1 for the syntax of method specification denoted by the nonterminal *methodSpec*. See also Appendix B for the nonterminal *identifier*.

\[
\begin{align*}
typeSpec & ::= \text{type typeId} \\
& \quad \text{supertypes type}^+, \\
& \quad \text{usedTrait traitId (tToSMapping)} \\
& \quad \text{methodSpec}^*
\end{align*}
\]

\[
\begin{align*}
tToSMapping & ::= \{\text{type for sortId}\}^+, \\
type & ::= \text{typeId} \\
typeId & ::= \text{identifier} \\
sortId & ::= \text{identifier}
\end{align*}
\]

Checking

A type specification must satisfy the following conditions to be syntactically well-formed:

1. The specified type appears in the type-to-sort mapping.

2. Each sort identifier appearing in the type-to-sort mapping is defined in the used trait.

3. Each method specification is well-formed (see Section 2.1).

Notes

1. Type specifications are constructed by other type specifications. In the above example, specification *IntegerSet* has all the instance method specifications of
specification `ObjectWithEqual` since type `IntegerSet` is specified as a subtype of type `ObjectWithEqual`. See Section 2.2.1 for more details.

2. Types are often recursively defined. For example, in the following instance method specification of type `IntegerSet`, `IntegerSet` is used to specify the type of the input formal `s`.

```plaintext
isSubsetOf: s <: IntegerSet
  returns  b <: Boolean
  modifies nothing
  ensures b = forall (i: Int) [include(s.pre,i) =>
   include(self.pre,i)]
```

### 2.2.1 Subtyping

The `supertypes` clause in a type specification defines a relation, called `subtype relation`, among all the specified types. The clause lists `direct supertypes` of the specified type — a type can have more than one direct supertype. These types, along with their direct supertypes, their direct supertypes’ direct supertypes, and so on, are called `strict supertypes` of the specified type. A type and its strict supertypes are called `supertypes` of the type. We use `subtype` for the inverse relationship; S is a subtype of T whenever T is a supertype of S. The subtype relation is reflexive, transitive, and antisymmetric, thereby, forming a partial order on all types. The single type `Object` is a ultimate supertype of all types; it specifies properties that hold for all objects. If S is specified to be a subtype of T, then S `inherits` all the instance method specifications of T (unless specialized\(^9\)). Only additional or changed methods

\(^9\)A method specification is said to be `specialized` in a subtype if the subtype specifies a method with the same selector.
need be specified in the specification of S. Specifying S in terms of its difference from T leads to a shorter specification, and a specification that is easier to maintain. To give a proper meaning to an inherited method, all operators used in that method specification must be properly defined in the used trait of the inheriting type. This is a specifier’s responsibility\(^\text{10}\), which can be easily done using \texttt{includes} clauses (see Section 1.1.2), or by appropriately defining those operators in the used trait.

Checking

A type is a description of object behavior. This description asserts that every object of type S is also logically of type T if S is specified to be a subtype of T. Any property that holds for T objects would also hold for S objects; messages understood by T objects will be understood by S objects and will have a similar effect. To provide this property, the subtype relationships must satisfy certain semantic constraints (called \textit{simulation relationships} in [31, 32]). However, these semantic constraints cannot be easily enforced by machines, so we adopt the traditional syntactic constraints, called \textit{syntactic subtyping rules}, listed in item 2 below. The syntactic subtyping rules are weaker than the semantic constraints. The following rules are applied to subtyping and inheritance:

1. The specified type can not appear in the \texttt{supertypes} clause. Only strict supertypes are listed without repetition.

\(^{10}\)Alternatively, we may assume that a subtype inherits both the instance methods and the used traits of its direct supertypes. However, this rule enforces the traits for a subtype to be specified in terms of those of its direct supertypes – i.e., subtype hierarchy in the interface level will be similar to the inclusion relationships (by \texttt{includes} clause) of traits in the shared level.
2. For each of direct supertypes \( T \) of a specified type \( S \), if an instance method \( M \) (specified in \( T \)) is specialized in \( S \) (i.e., has the same selector), the following antimonotonic subtyping rules holds [4]:

- The return type of \( M \) in \( S \) is a subtype of that of \( M \) in \( T \).
- For each input arguments of \( M \), its type in \( T \) is a subtype of that in \( S \).

3. An instance method specified by more than one direct supertypes with the same selector, is not inherited by a subtype.\textsuperscript{11}

This subtyping rules together with inheritance guarantees that a message understood by objects of a type is also understood by objects of its subtypes. However, the effects of the message sending are not guaranteed to be the same by these syntactic rules.

\textbf{2.2.2 Mutation}

An object whose values can change is said to be \textit{mutable}; one whose value cannot change is said to be \textit{immutable}. A type is \textit{mutable} if objects of that type are mutable, otherwise it is \textit{immutable}. For example, integers and booleans are immutable objects while sets are mutable objects. So type \texttt{Integer} and \texttt{Boolean} are immutable types while type \texttt{Set} is mutable. In Larch/Smalltalk, we specify a type to be mutable or immutable with a \textbf{mutation} clause. If this clause is omitted, the specified type is assumed to be mutable by default. We adopt this rule simply because most of the types specified are mutable.

\textsuperscript{11}Instead of this, we can adopt similar rule saying that an instance method with the same selector cannot be specified by more than one direct supertypes. Another alternative is to disjoin the pre-conditions, conjoin the post-conditions and intersect the modifies lists of all the method specifications with the same selector.
Example

```plaintext
type IntegerSet
  mutation true
  supertypes ObjectWithEqual
  usedTrait SetOfE(IntegerSet for S, Integer for E)
```

In the above example, type `IntegerSet` is explicitly specified to be mutable.

Syntax

```plaintext
typeSpec ::= type typeId
  mutation mutationFlag

  ......

mutationFlag ::= true | false
```

Checking

1. If a type is specified to be mutable, its specification must have at least one method specification with a `modifies` clause that includes the object identifier `self`.

2. No supertype of an immutable type can be mutable.

2.2.3 Invariants

In our specifications, the abstract values of a specified type are given by the used trait in the shared level. These values in the shared level are purely mathematical
and specified without concern about implementation details, thereby, allowing traits to be shared by many type specifications. However, at the interface level, one must cope with implementation details and restrictions. This can be done by an invariant clause in type specifications, which list a predicate that must be preserved by all the objects of that type. In other words, an invariant is a property that must hold for all objects of a specified type. Thus an invariant clause can be used to restrict the values (domain) of a type to a subset of the values defined by traits in the shared level. The invariant must be preserved by each method specification that mutates the receiver, which is specifiers’ responsibility. If no invariant is specified, “true” is assumed by default.

Example

```plaintext
type Natural
  mutation false
  supertypes Number
  usedTrait NatTrait(Natural for Nat)
  invariant forall (n: Natural) [n ≤ 2^{32}]

where 2^{32} ≡ succ(succ(· · · succ(zero) · ·))

the result of applying succ to zero 2^{32} times.
```

Trait NatTrait shown in Fig. 2.6 defines an infinite set of numbers, i.e., all the natural numbers, with simple operations like + and <. Since machines can not represent infinitely many numbers, type Natural restricts, by invariant clause, its domain to a finite set, all natural numbers less than 2^{32}, a subset of the abstract domain specified by trait NatTrait. The invariant says that every instance of type Natural must be less than 2^{32}, which allows a 32-bit fixed-point representation of
NatTrait(Nat): trait
    includes Bool
    introduces
        zero: → Nat
        succ: Nat → Nat
        +: Nat, Nat → Nat
        <: Nat, Nat → Bool
    asserts
        Nat generated by zero, succ
    forall i, j: Nat
        one == succ(zero)
        i + succ(j) == succ(i + j)
        i < zero == false
        zero < succ(i) == true
        succ(i) < succ(j) == i < j
        i + j == j + i

Figure 2.6: Trait NatTrait

numbers.

Syntax

The nonterminal assertion, which stands for a first order predicate, is defined in Section 2.1.1. See Appendix B for the nonterminals type and identifier.

\[
typeSpec ::= type \typeId \\
\ldots \\
\text{invariant} \quad \text{invariant} \\
\text{methodSpec}^*\]
\[
\begin{align*}
\text{invariant} & \quad ::= \quad \text{true} \mid \text{forall} (\text{varId: type}) [\text{assertion}] \\
\text{varId} & \quad ::= \quad \text{identifier}
\end{align*}
\]

Checking

The assertion of the \textit{invariant} clause must be well-formed in the sense defined in Section 2.1.1.

\section{Parameterized Type Specifications}

In Larch/Smalltalk, type specifications may have parameters, called \textit{type parameters}. Type specifications with parameters are called \textit{parameterized type specifications}. A parameterized type specification is a generalization of a simple type specification, and specifies a family of related types, not just a single type. So a parameterized type specification can be thought of as a specification generator. A particular member of this family has as its specification the generator with all the parameters replaced by actual types. This newly instantiated type is called \textit{parameterized type} and is named by giving the generator name followed by one or more parameters in parentheses, e.g., $\text{Set(Integer)}$\footnote{As with simple types, $\text{Set(Integer)}$ is used to denote both the specified type and its specification.} is a parameterized type with a generator \texttt{Set} and an actual parameter \texttt{Integer}.

The parameterized specification, $\text{Set}$ (see Fig. 2.7), specifies a family of sets; set-of-integers, set-of-characters, set-of-strings, and so on. The \texttt{parameters} clause declares a type parameter, \texttt{ELEMTYPE}, that stands for the element type of sets and must be a subtype of type \texttt{ObjectWithEqual}. For any subtype of \texttt{ObjectWithEqual}, $T$, the
type Set
parameters ELEM_TYPE < ObjectWithEqual
supertypes Collection(ELEM_TYPE)
usedTrait SetOfE(Set(ELEM_TYPE) for S, ELEM_TYPE for E)

isIn: e :< ELEM_TYPE
  returns b :< Boolean
  ensures b = include(self.pre,e)

......

Figure 2.7: A parameterized type specification Set

objects of type Set(T) are sets whose elements are of type T, and its specification is Set with all the occurrence of parameter ELEM_TYPE replaced by T. For example, Set(Integer) is the type of set-of-integers and its objects are sets whose elements are of type Integer. The specification of type Set(Integer) is the specification Set with all the occurrence of ELEM_TYPE replaced by Integer. By defining a parameterized type specification, Set, once and for all, we produce an unlimited number of type specifications; Set(Integer), Set(Character), Set(String), and so forth.

Syntax

See Section 2.2 for typeSpec in detail. See also Appendix B for identifier.

\begin{align*}
  typeSpec &::= \text{type typeId} \\
  & \quad \text{parameters paramList} \\
  & \quad \ldots \\
  \text{paramList} &::= \{\text{paramId} [<_\text{type}]\}^+,
\end{align*}
The **parameters** clause lists a sequence of type parameters. The range of a type parameter, the upper bound of instanciation, may be restricted by including an optional range type declaration. By default, the range type is assumed to be **Object**, the ultimate supertype of all types. A type parameter can only be instanciated to only those types that are subtypes of its range type.

**Checking**

1. Type parameters listed in a **parameters** clause must be distinct.

2. The range type of a type parameter, if specified, must be a manifest type, i.e., a type whose name does not include any type parameters. For example, **Set(Integer)** is a manifest type while **Set(ELEM_TYPE)** is not.

**Notes**

1. The scope of a type parameter declaration is limited to the generator, including all its method specifications.

2. A type parameter acts as a named constant, thus, can be used anywhere a simple type is allowed except for in the range declaration of parameters where a manifest type is expected.
3. A parameterized type like Set(Integer) can be used anywhere a simple type can, both in method specifications and in type specifications.

4. The ability to restrict the range of a type parameter, i.e., to restrict the parameter to subtypes of a particular type, turns out to be very useful in specifications. We can assume that certain property holds in a parameter (i.e., some operations are defined). In the Set example, we assumed that equality of two elements can be tested by declaring ELEM_TYPE to be a subtype of type ObjectWithEqual that describes objects that can respond to the equality testing messages (=, ≠). See [5] for the notion of bounded quantification.

5. By convention, all capital letters are used for type parameters.

2.3.1 Subtyping

Since a parameterized type specification defines a family of types, we can specify subtype relationships in several different ways. For example, all the members of the family may be specified to have a common direct supertype, or each member to have different direct supertype. Below are listed different ways of specifying the direct supertypes of a parameterized type.

1. All the family members can be direct subtypes of a common simple type. For example,

```plaintext
type Set
  parameters ELEM_TYPE < ObjectWithEqual
  supertypes ObjectWithEqual
  .......
```
would make each type `Set(T)` (where `T` is an arbitrary subtype of type `ObjectWithEqual`) a direct subtype of type `ObjectWithEqual`.

2. By using type parameters, family members can be given different direct supertypes. For example,

```java
type Set
  parameters ELEM_TYPE < ObjectWithEqual
  supertypes Collection(ELEM_TYPE)
  .......
```

would make each `Set(T)` a direct subtype of the corresponding `Collection(T)`.

3. All the family members can be direct subtypes of a parameterized type. For example,

```java
type Set
  parameters ELEM_TYPE < ObjectWithEqual
  supertypes Collection(Object)
  .......
```

would make each `Set(T)` a direct subtype of type `Collection(Object)`.

4. Any combination of above three can be used. For example,

```java
type Set
  parameters ELEM_TYPE < ObjectWithEqual
  supertypes Collection(Object), Collection(ELEM_TYPE), ObjectWithEqual
  .......
```
would make type \texttt{Set(Integer)} have three different direct supertypes: \texttt{Collection(Object)}, \texttt{Collection(Integer)}, and \texttt{ObjectWithEqual}.

Checking

For a parameterized type specification to be correct, the subtype relations imposed by the \texttt{supertypes} clause must not violate the subtyping rules given in Section 2.2.1.

2.4 Block Specifications

A Smalltalk \texttt{block} is a closure, that is to say, it contains a parameterized code and an environment to look up variables that are referred in the code, but are not parameters. In Smalltalk terms, a block is an object representing a sequence of actions to be taken at a later time, upon receiving messages with selectors such as \texttt{value}, \texttt{value:}, \texttt{value:value:}, and so on [13]. In Smalltalk, all blocks are instances of the same class. However, it is appropriate to send the message \texttt{value} only to a block taking no arguments, the message \texttt{value:} to a block taking one argument, the message \texttt{value:value:} to a block taking two arguments, and so on. This leads us to specify blocks by a family of types according to their numbers of input arguments [3]. Since the same block can take and return objects of different types, it would be appropriate to parameterize specifications of blocks according to their input and return types.

One approach to the block type specifications is to view blocks as mappings on execution states. The evaluation of a block, therefore, is abstracted as a transition of execution states. The return value of a block evaluation is the value in a new state
given by the mapping. A special object is assumed to store the return value. Each instance of a block type will have a different mapping according to its behavior, that is to say, the mapping is determined by its actions to be taken, upon receiving an evaluation message (e.g., value, value). However, the precise mapping is not given in the specification. It is assumed to be given by the system when a new object (block) is created.

```
type Block1
    parameters ARGTYPE -> RETTYPE
    mutation false
    supertypes Object
    usedTrait Block1(Block1(ARGTYPE -> RETTYPE) for Blck)

value: i :< ARGTYPE
    returns o :< RETTYPE
    ensures o = eval(self.pre, i)
```

The above example shows a specification of blocks taking only one argument, type Block1. Type parameters ARGTYPE and RETTYPE stand for types of the input argument and the output argument respectively; note that RETTYPE is preceded by a right arrow (->). It specifies only one instance method, value; which takes an object of type ARGTYPE and returns an object of type RETTYPE, the result of evaluation of the receiver, an instance of the specified block. At the shared level (see trait Blockl in Fig. 2.8), an execution state is modeled as a mapping from objects to their values. A state should include both the explicit arguments and implicit arguments (e.g., global variables\textsuperscript{13}). A block is abstracted as a pair of execution states.

\textsuperscript{13}A global variable is a variables that is referred in a block, but is not a block argument.
and mapping on these states. A state in the pair denotes the state on which a block is 
evaluated; that is to say, the evaluation of a block is modeled as getting a new state 
from the mapping, based on the state and the input argument, and returning the 
value of object RETOBJ in the new state. Note that RETOBJ is a constant of sort 
OBJ denoting the object that stores the return value, and ARG_OBJ is a constant 
that denotes the input argument.

In the above approach, one cannot specifies blocks that are used for side-effects 
since the value: method asserts that no objects (including globals) are allowed to be 
mutated. This is because possible globals are unknown until an instance (block) is 
created. Further work is expected to extend current syntax or to specify blocks in 
different ways.

Syntax

The syntax for parameterized type specifications is extended to allow one to 
specify blocks as follows:

\[
\begin{align*}
typeSpec & ::= \text{type} \; typeId \\
\text{parameters} & \{\text{paramList [-> paramDecl]} \\
& \mid -\text{-> paramDecl}\} \\
\end{align*}
\]

\[
\begin{align*}
\text{paramList} & ::= \text{paramDecl}^+ \\
\text{paramDecl} & ::= \text{paramId} \; [< \text{type}] \\
\text{type} & ::= \text{simpleType} \mid \text{param Type} \\
\text{simpleType} & ::= \text{typeId} \\
\text{param Type} & ::= \text{typeId}(\text{type}^+ \mid [-> \text{type}]) \mid \text{typeId}(-\text{-> type}) \\
\text{typeId} & ::= \text{identifier} \\
\text{paramId} & ::= \text{identifier}
\end{align*}
\]
Block1: trait
  includes State(stateEval for eval), Mapping(St for D, St for R)
  introduces
    new: M, St → Blck
    map: Blck → M
    state: Blck → St
    evalForState: Blck, Val → St
    eval: Blck, Val → Val
  asserts
    Blck generated by new
    forall b: Blck, m: M, s: S, v: Val
      state(new(m,s)) == s
      map(new(m,s)) = m
      evalForState(new(m,s),v) == range(m, bind(s,ARGOBJ,v))
      eval(b,v) == stateEval(evalForState(b,v), RETOBJ)

% see trait Mapping for bind and range
% see Appendix A for traits State and Mapping

Figure 2.8: Trait Block1 specifying blocks taking one argument.

2.4.1 Subtyping

In addition to subtyping relations given explicitly by the supertypes clause\(^{14}\), a parameterized block type S is a subtype of type T if and only if:

\(^{14}\text{It is doubtful to explicitly specify subtype relationships among block types using the supertypes clause. However, it is left to make the syntax orthogonal with respect to simple parameterized type specifications.}\)
1. Both S and T are instantiations of the same specification; this implies that objects of S and objects of T take the same number of arguments.

2. The return type of T is a subtype of that of S.

3. For each argument type of S, it is a subtype of the corresponding argument type of T.

For example, Block1(Integer -> Natural) is a subtype of Block1(Natural -> Integer) if type Natural is a subtype of type Integer.

2.5 Summary

In this chapter we described Larch/Smalltalk and its informal semantics with examples. A type specification is the basic unit of specifications in our language, and consists of a type name, direct supertypes, a used trait, a type-to-sort mapping, and a set of method specifications. Two kinds of methods are specified; instance methods and meta methods which correspond to instance methods and class methods in Smalltalk. A specified type inherits (unless specialize) all the instance method specifications of its supertypes. A method is specified by a pair of predicates called pre- and post-conditions. An object can be qualified to refer to its initial and final value since assertions are made with respect to two states, the initial state and the final state of a method. Special clauses were introduced to assert creation of new objects and modification of existing objects.

A type specification may have type parameters. A parameterized type specification specifies a family of related types, not just a single type; it was shown how to specify a parameterized type Set. Finally, Smalltalk blocks are described by a
family of types, according to the numbers of input arguments, in an extended form of parameterized type specifications.
CHAPTER 3. SUPPORT TOOLS FOR LARCH/SMALLTALK

Like Smalltalk programs, specifications in Larch/Smalltalk are expected to be developed in an interactive graphical environment with an editor, syntax- and sort-checker, specifications manager, printing utilities, and a window system. A system type browser and a system trait browser, which are integrated in Smalltalk-80 system, provide this developing environment. A system type browser (see Fig. 3.9) allows specifications in Larch/Smalltalk to be entered, modified, browsed, and syntax-and sort-checked; a system trait browser (see Fig. 3.17) does similar work on the shared components of specifications, i.e., traits specified in the Larch Shared Language. These browsers were implemented in Version 2.5 of the Smalltalk-80. In the following two sections, each of these is described in detail. For full explanation of the Smalltalk’s graphical user interface including windows and mice, see [12] and [29].

3.1 System Type Browser

A system type browser is created by selecting type browser entry from the Smalltalk system menu (see Chapter 2 of [12] and Chapter 3 of [29] for the system menu). A system type browser is divided into eight scrollable panes (or subwindows) and two switch panes labeled instance and meta (see Fig. 3.9). The top four and bottom three panes are called list panes, while the center one is called a text pane.
Figure 3.9: A system type browser

A list pane contains a fixed list of items that can be selectable but cannot be edited directly. A list pane is scrollable, i.e., to view all the available items, it may be necessary to scroll through the contents of the pane. Text within a text pane may be scrolled, selected, and edited. A browser can be closed, collapsed, moved, and framed using the blue button menu\(^1\).

Each subwindow except for switches has a menu, called yellow button menu, accessible through the yellow button. This menu contains operations to be applied within the context of the currently selected items. Fig. C.23 on page 82 shows the typical yellow button menus associated with each of the browser panes. The actual

\(^1\)A typical Smalltalk system uses a three-buttoned mouse; the left button is called red button, the center one is yellow button, and the right one is blue button. The red button is used to select information, the yellow to activate a menu (yellow button menu) for editing the contents of a window, and the blue to activate a menu (blue button menu) for manipulating the window itself. For more detail, see Chapter 2 of [12] and Chapter 3 of [29].
entries in these menus may depend on the selections made within the list panes of
the browser at the time the menu is activated.

Related type specifications are grouped together into type categories, and related
method specifications within individual types are grouped into message categories.
The four upper panes from left to right list type categories, type names, message cat-
egories, and message selectors. The switch panes below the messages category pane
determines whether instance messages or meta messages are displayed in the mes-
age categories and message selectors panes; these are on-off switches, i.e., selecting
instance deselects meta and vice versa. The bottom three list panes show names of
implementing classes, direct subtypes, and direct supertypes of the currently selected
type in the type names pane. Selections are made from the list panes and switch
panes using the red button. When a browser window is deactivated or collapsed, the
current selection from the menus are remembered and restored when the browser is
reactivated or framed at some later time.

3.1.1 Viewing existing specifications

A type browser provides access to all the relevant information concerning type
specifications; the used trait, type hierarchy, instance and meta protocols, method
specifications, and so on. This information is usually displayed in the text pane
of the browser by selecting entries from the list panes and from the various pane
menus. What is displayed in a list pane is dependent on the selections previously
made in the other list panes; each of the upper four list panes is dependent on its
neighboring list panes (to the left) and the three bottom list panes depend on the type
names pane. In Fig. 3.9, for example, the type category LarchST-Compilers-Objects
is selected. Hence, the type names displayed in the type names pane are the types in this category. The type \texttt{QualifiedIdentifierNode} is selected and this, together with the fact that \texttt{instance} switch is on, determines that instance method categories of type \texttt{QualifiedIdentifierNode} will be displayed in the message categories pane. The message category \texttt{access} is selected, thereby, the message selectors for instance methods in the category \texttt{access} in the type \texttt{QualifiedIdentifierNode} are displayed in the message selectors pane. Finally, the selector \texttt{qualifier:} is selected, causing the specification for this method to be displayed in the text pane. The bottom three list panes respectively show the names of implementing classes, direct subtypes, and direct supertypes of type \texttt{QualifiedIdentifierNode}, the type selected in the type names pane.

\textbf{3.1.1.1 Finding type specifications} Although types are organized in a hierarchy, they are displayed in a different manner in a browser, i.e., they are categorized into sets of functionally related types. So to view a type, one must know its category, or scan through the categories one by one. However, it is not likely that one knows the categories of all the types in the system, and considerable time can be spent on seeking through the type categories one by one. A fast way to find a type is to select \texttt{find type} entry in the yellow button menu (see Fig. C.23 on page 82) of the type categories pane. The browser will ask the name of type to be located. When a type name is entered, the requested type and its category will be selected in the type names pane and in the type categories pane respectively. Typing a pattern string using ‘*’ as a wild character, will bring up a list of type names matching the pattern. Selecting one causes the browser to position itself at that type.
3.1.1.2 Viewing type specifications To display the definition of a type, first select the type from the type names pane, then select definition from the yellow button menu (see Fig. C.23 on page 82) of the same pane. The definition of the type will appear in the text pane. Fig. 3.10 shows the specification of type QualifiedIdentifierNode. The definition shows its mutability, direct supertypes, the name of used trait, and type-to-sort mapping. Note also that selecting a type in the type names pane causes its implementing classes, direct subtypes, and direct supertypes to be listed in the corresponding panes. Class LSTQualifiedIdentifierNode together with class SpecNodeBuilder implement type QualifiedIdentifierNode. Type QualifiedIdentifierNode has one direct supertype, IdentifierNode, and no subtype.

3.1.1.3 Viewing the type hierarchy To view the subtype hierarchy for a type, first select the type from the type names pane, then choose hierarchy from
the yellow button menu of the pane. The type hierarchy (in the indented form) is displayed in the text pane and shows the subtype chain above and below the type.

3.1.1.4 Viewing the protocol supported by a type To display the instance or meta protocol of a type, first bring the type in the text pane (see Section 3.1.1.2), then select protocols from the yellow button menu of the type names pane. The instance or meta protocol is displayed in the text pane according to the current selection of instance-meta switch. For example, if one chooses to display instance protocol of type QualifiedIdentifierNode, the following will appear in the text pane:

('qualification' objectQualified postQualified preQualified
 valueQualified)
('access' qualifier qualifier:)

Each entry describes the protocol associated with each message category; the first item in single quotes is a category name, and all the following are the names of methods in that category. The displayed entries can be edited and accepted, for example, to change protocol categories, or to reorder them.

3.1.1.5 Viewing method specifications To display the specification for a method in the browser, proceed with the following steps:

1. Select the type category from the type categories pane.

2. Select the type from the type names pane.

3. Set the instance-meta switches to either instance or meta.
Figure 3.11: Message list generated by using `find method` for type `AssertionNode`

4. Select the method category from the method categories pane.

5. Select the message selector from the message selector pane.

Fig. 3.9 shows the instance method specification with selector `qualifier`: in type `QualifiedIdentifierNode`.

### 3.1.1.6 Finding method specifications

If one does not know the category of a method specification or unsure of its spelling, the `find method` entry in the yellow button menu of the type names pane may be used. Selecting this entry causes the browser to display a list of messages specified by the currently selected type. For example, in Fig. 3.11 the messages specified by type `QualifiedIdentifierNode` are displayed. To view a particular method specification, select the desired message selector from the list.
3.1.2 Modifying and adding specifications

3.1.2.1 Modifying and adding method specifications  Existing method specifications can be modified as follows:

1. Bring the method specification to be modified to the text pane (see Section 3.1.1).

2. Edit the specification using the operations available from the yellow button menu of the text pane (cut, copy, paste, again, undo, etc). Chapter 3 of [12] explains how to use this text editor (see also Section 3.4 of [29]).

3. Select accept from the same menu to store the modified version into the system.

On choosing accept the system does syntax- and sort-checking of the specification (see Chapter 2). If there is an error, it will be indicated by a high-lighted message (see Fig. 3.12); otherwise, this modified version will be kept by the system in place of the original one.

To add a new method specification to an existing type, select appropriate type category, type, and message category. In the text pane, will be displayed a method template (see Fig. 3.12). Edit this template and select accept as in the above.

3.1.2.2 Modifying and adding type specifications  To modify an existing type specification, display its definition in the text pane by making appropriate selections in the categories and type names panes. Edit the definition in the text pane and select accept in the yellow button menu. To add a new type specification into the system, select the category in the type categories pane in which the new type will be kept. Then, a type specification template will appear in the text pane. Edit this template and select accept from the yellow button menu.
3.1.2.3 Adding new type (message) categories  A new type category is added to the system by choosing add category in the yellow button menu of the type categories pane. A prompter window will appear asking a category to be added. Simply type in the name of category to be added. A new message category is added as the same way by selecting add category from the yellow button menu of the message categories pane and typing in a category to be added.

3.1.3 Specialized browsers

3.1.3.1 Browsing by category, type, message category, and message Category, type, message category, and message browsers are browsers that limit access only to the specified category, type, message category, and message. They can be thought of as degenerated system type browsers.

A category browser provides access only to the types in a specified category.
Figure 3.13: A type category browser (left) and a type browser (right)

Except for this, it provides the same functionality as a system type browser. A category browser is opened by selecting spawn from the yellow button menu (see Fig. C.23 on page 82) of the type categories pane. Fig. 3.13 shows a category browser on the type category LarchST-Compilers-Objects.

A type browser allows access to only a specified type. Except for this, it provides the same functionality as a category browser. A type browser can be opened by choosing the spawn entry from the yellow button menu of the type names pane. Fig. 3.13 shows a type browser opened on type QualifiedIdentifierNode.

A message category browser and a message browser are two other seldom used browsers. A message category browser limits access to only the messages in the specified category; a message browser shows only a single method specification with the specified selector. These browser are created similar way by choosing the spawn entry from the yellow button menus of the corresponding panes; the message categories pane and the message pane. Fig. 3.14 shows a message category browser opened on the message category access in type QualifiedIdentifierNode and a message browser on the selector qualifier.
3.1.3.2 Browsing through subtype chain  Since a type inherits instance method specifications (unless overridden) from its supertypes, it should not be viewed in isolation, that is to say, one must view a type in its subtype chain to get the full picture. A type hierarchy browser expedites this viewing of type specifications. It is organized around type hierarchies rather than around categories. However, in structure and functionality, it is similar to a category browser, except that the types displayed are only the supertypes and subtypes of the selected type. A type hierarchy browser is opened by choosing spawn hierarchy entry from the yellow button menu (see Fig. C.23 on page 82) of the type names pane after the type to be browsed is selected. Fig. 3.15 shows a type hierarchy browser opened on type QualifiedIdentifierNode. The type names pane contains type QualifiedIdentifierNode, supertypes IdentifierNode, AssertionNode, SpecNode, ObjectWithEqual, and Object (Type QualifiedIdentifierNode has no subtype).

3.1.3.3 Browsing sets of method specifications and methods  Message-set browsers allow one to browse a collection of method specifications and methods (in Smalltalk) with some common characteristics; e.g., method specifications with
Figure 3.15: A type hierarchy browser opened on type `QualifiedIdentifierNode`

the same selector, and methods that implement a particular method specification.

It is often useful to browse through the method specifications with a particular selector, for example, to see how a method is specialized in different subtypes. A message-set browser on the set of method specifications with the same selector can be opened by selecting the `specifiers` entry from the yellow button menu (see Fig. C.23 on page 82) of the message pane after the selector to be examined is chosen. Fig. 3.16 (bottom) shows all the method specifications with selector `name`. In the list pane of the browser, a message name is preceded by the type in which it is specified.

The ability to view all the implementations (methods in Smalltalk) for a particular method specification (i.e., method level specifications-to-implementation link) is extremely useful in incrementally and simultaneously specifying and implementing Smalltalk programs. A message-set browser (of Smalltalk) on the implementing methods for a particular specification can be created by selecting the `implementors`
entry from the yellow button menu of the message selector pane. Fig. 3.16 shows the method specification with selector name in type SpecNode (in the system type browser) and its implementing methods in Smalltalk (right).

3.1.4 Viewing the shared component

To understand the meaning of a type specification in Larch/Smalltalk, one need to see the shared component, i.e., the used trait specified in the Larch Shared Language. This trait can be located and viewed by a system trait browser (see Section 3.2), which is similar to a system type browser. However, a system type browser provide a simple way to browse the used trait of the currently selected type in the type names pane. The yellow button menus of the type names pane, the message categories pane, and the message pane contain the entry spawn trait (see Fig. C.23 on page 82), which opens a trait browser (see Fig. 3.18 on page 65), a degenerated sys-
tem trait browser on the used trait of the currently selected type. The trait browser can be updated using the yellow button menus of its text pane.

3.1.5 Browsing implementing classes and direct subtypes (supertypes)

The bottom three panes of a type browser list (from left to right) the implementing classes, the direct subtypes, and the direct supertypes of the currently selected type in the type names pane. Using the operations available in the yellow button menu of the implementing classes pane, one can add, browse, remove implementing classes of the selected type (see Section C.1.5 in Appendix C). One can also add, select, browse, and delete direct subtypes (supertypes) of currently selected type in the type names pane using the yellow button menu commands of the direct subtypes (supertypes) pane (see Section C.1.6 in Appendix C).

3.1.6 Saving specifications

3.1.6.1 Filing out Method and type specifications can be saved to an external file in a format that can be read subsequently back into the system. This filing-out can be done at four levels: type categories, type specifications, message categories, and individual method specifications. The four top yellow button pane menus (see Fig. C.23 on page 82) in a system type browser have the file out entries corresponding to the four levels of output (from left to right). File names, which can be changed by the user, are automatically generated according to the conventions shown in Table 3.3.

Files are written out in a special standard format (similar to that of Smalltalk code; see Chapter 22 of [12] and Section 4.7 of [29]) so that they can be read back
### Table 3.3: File name conventions

<table>
<thead>
<tr>
<th>Information</th>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>categoryName.lst</td>
</tr>
<tr>
<td>type specification</td>
<td>typeName.lst</td>
</tr>
<tr>
<td>message category</td>
<td>typeName-messageCategoryName.lst</td>
</tr>
<tr>
<td>method specification</td>
<td>typeName-messageSelectorName.lst</td>
</tr>
</tbody>
</table>

into system using the Smalltalk `fileIn` message (see below), the inverse of `file out`.

#### 3.1.6.2 Printing
The entry `print out` also appears in each of the upper four pane menus of a system type browser. It will write out specifications to an external file in a pretty printed form suitable for human reading (PostScript format in the current implementation). This file cannot be subsequently read back into system using `fileIn`. By comparison, the form produced by `file out` is designed to be readable primarily by the system. As in `file out`, specifications can be printed out at the four levels with same file name conventions except that `.`ps` (instead of `.lst`) is used as a file name extension.

#### 3.1.6.3 Filing in
Files produced by `file out` can be subsequently read back into the system with the Smalltalk `fileIn` message using an expression of the following form (see Section 4.7 of [29]).

```
(FileStream oldFileNamed: fileName) fileIn
```

Or using the the Smalltalk `file list browser` which can be opened by selecting `file list` from the Smalltalk system menu (see Chapter 22 of [12]). Using this browser, one can file-in specifications as the same way as one does with Smalltalk code files.
Note that when new specifications are added to the system by filing-in, system type browsers that are already open will not automatically contain these specifications. Select **update** from the yellow button menu in the type categories pane to update the browsers.

### 3.2 System Trait Browser

A system trait browser is a browser used to browse traits specified in the LSL, and is opened by selecting the **trait browser** entry from the Smalltalk system menu. A system trait browser consists of three scrollable panes; trait categories pane, trait names pane, and text pane (see Fig. 3.17). As in a system type browser, related traits are grouped together into categories, which are shown in the categories pane. All the traits in the category selected in the categories pane are listed in the trait names pane. The text pane shows the definition of the trait selected in the trait names pane. A trait browser can be closed, collapsed, moved, and framed by the blue button menu.

Using the yellow button menu operations available on each panes (see Fig. C.24 on page 86 and Section C.2), one can find, view, add, modify, file-out, print, and remove traits from the system in the similar way as in a system type browser. One can also create specialized browsers, **category browser** and **trait browser**, which limit access to only the traits in the specified category and a single trait respectively (see Fig. 3.18). Note also that the text pane has the same operations available from the yellow button menu as that of a system type browser. Appendix C gives a brief explanation of each yellow button command available in the categories pane and in the trait names panes.
Figure 3.17: A system trait browser

Figure 3.18: A trait category browser (left) and a trait browser (right)
CHAPTER 4. CONCLUSIONS

We designed a Larch-style two-tiered specification language for Smalltalk, called Larch/Smalltalk, that supports subtyping as found in object-oriented programming languages. As a Larch interface language, it maintains the benefits of two-tiered approach (e.g., separation of concerns, division of effort, and reusability) while supporting the modularity of specifications similar to those in object-oriented programming languages. We implemented supporting tools (browsers) for Larch/Smalltalk, which are integrated in the Smalltalk-80 programming environment. Using these tools, we specified a portion of Smalltalk system classes in Larch/Smalltalk. In addition, we also specified a part of the tools themselves, a parser for Larch/Smalltalk.

The design of specification languages for object-oriented programming languages and implementation of supporting tools presented here, however, have only expository status and represents only a small proportion of what remains to be done. Especially in the following areas, we expect future work will be continued in connection with formal verification techniques: (1) to define a rigorous formal semantics of Larch/Smalltalk, (2) to enrich the expressive power of Larch/Smalltalk, (3) to develop better approaches to the specifications of block types, (4) to develop techniques of verifying and reasoning about Smalltalk programs specified in Larch/Smalltalk, (5) to develop a library of specifications that can be reused or refined (by subtypes).
As discussed in Chapter 1, there has been much effort on designing object-oriented specification languages and on ways of specifying programs in object-oriented programming languages. Below are listed a few contributions of our work to this effort:

1. Larch/Smalltalk is a new specification language for Smalltalk. So far, there has been no specification language targeted to Smalltalk programming language.

2. Larch/Smalltalk is an object-oriented specification language based on types and subtypes mechanism, not on classes and subclasses. Specifications of a supertype are inherited by a subtype of that type, allowing reuse of specifications in the interface level\(^{16}\). Subtype relations satisfy certain constraints, called syntactic subtyping rules, so that an object of subtype can be used in place of an object of its supertype. Subtype relationships also help to modularize specifications according to their conceptual relationships. Since our specification language is based on types, it is well suited to the existing typed Smalltalk programming languages [3, 14], extensions of pure Smalltalk.

3. We introduced two-tiered approach to specifications to object-oriented programming languages. Our specification language and Larch/Avalon [10] are the only Larch-style two-tiered specification languages for object-oriented programming languages.

4. We tried to integrate formal specifications in the Smalltalk-80 programming environment by providing such support tools as type browsers and trait browsers.

\(^{16}\)Other Larch interface specifications languages like Larch/Pascal and Larch/CLU, have no way of reusing specifications in the interface level. Only traits (in the shared level) specified in the LSL can be shared by different interface specifications.
Using these browsers, specifications in Larch/Smalltalk are expected to be developed interactively and incrementally while simultaneously implementing Smalltalk programs in Smalltalk code browsers. This environment encourages specifications to be used productively in the programming process.

5. Existing object-oriented reasoning and verifying techniques [31, 32] can be easily adapted to reason about Smalltalk programs specified in Larch/Smalltalk.
BIBLIOGRAPHY


APPENDIX A

Trait **Boolean** and **Equality** are built in Larch/Smalltalk, that is to say, they are implicitly included into the shared component of each specification. Trait **Boolean** defines normal boolean operators (not, &, |, =>, <=); trait **Equality** defines two heavily overloaded equality operators (=, ~=).

Equality: trait
    includes Boolean
    introduces
        =: S, S → Bool
        ~=: S, S → Bool
    assert
        S partitioned by =
        forall x, y, z: S
            x = x == true
            x = y == y = x
            (x = y & y = z) => x = z == true
            x ~= y == not(x = y)

Figure A.19: Built-in trait Equality

Note that the sort S is renamed (overloaded) for each sort that appears in the shared component of a specification.
Boolean: \textbf{trait}
  \textbf{introduces}
    true: \to \text{Bool}
    false: \to \text{Bool}
    not: \text{Bool} \to \text{Bool}
    &: \text{Bool, Bool} \to \text{Bool}
    |: \text{Bool, Bool} \to \text{Bool}
    =>: \text{Bool, Bool} \to \text{Bool}
    <=>: \text{Bool, Bool} \to \text{Bool}
  \textbf{assert}
    \text{Bool generated by} \text{true, false}
    \textbf{forall} b: \text{Bool}
      not(true) == false
      not(false) == true
      true \& b == b
      false \& b == false
      true \mid b == true
      false \mid b == b
      true => b == b
      false => b == false
      true <=> b == b
      false <=> b == not(b)

Figure A.20: Built-in trait \textbf{Boolean}
State: **trait**

includes Object, Value

introduces

empty: \( \rightarrow \) St
allocate: St, Obj \( \rightarrow \) St
bind: St, Obj, Val \( \rightarrow \) St
isIn: St, Obj \( \rightarrow \) Bool
eval: St, Obj \( \rightarrow \) Val

assert

St generated by empty, allocate, bind
St partitioned by isIn, eval

forall s, s1: St, o, o1: Obj, v: Val

\[ \text{not}(\text{isIn}(\text{empty})) \]
\[ \text{isIn}(\text{allocate}(s,o),o1) = (o = o1) \mid \text{isIn}(s,o1) \]
\[ \text{isIn}(\text{bind}(s,o,v),o1) = (o = o1) \mid \text{isIn}(s,o1) \]
\[ \text{eval}(\text{allocate}(s,o),o1) = \text{eval}(s,o1) \]
\[ \text{eval}(\text{bind}(s,o,v),o1) = \]
\[ \text{if o = o1 then v else eval}(s,o1) \]

\%

used in trait Block1 in Fig. 2.8

---

Figure A.21: Trait State
Mapping: \texttt{trait}
\begin{itemize}
\item \texttt{introduces}
  \begin{itemize}
  \item empty: \rightarrow M
  \item assign: M, D, R \rightarrow M
  \item range: M, D \rightarrow R
  \item include: M, D \rightarrow \text{Bool}
  \item isEmpty: M \rightarrow \text{Bool}
  \end{itemize}
\end{itemize}
\texttt{assert}
\begin{itemize}
\item M \texttt{generated by} empty, assign
\item M \texttt{partitioned by} isEmpty, include, range
\item \texttt{forall} m: M, d, d1: D, r: R
  \begin{itemize}
  \item not(isIn(empty))
  \item isEmpty(assign(m,d,r))
  \item not(include(empty,d))
  \item include(assign(m,d,r),d1) == (d = d1) \mid \text{include}(m,d1)
  \item range(assign(m,d,r),d1) ==
    \begin{itemize}
    \item if d = d1 then r else range(m,d1)
    \end{itemize}
  \end{itemize}
\end{itemize}

\% used in trait Block1 in Fig. 2.8

Figure A.22: Trait \texttt{Mapping}
APPENDIX B

The syntax of Larch/Smalltalk is presented in an extended BNF with conventions shown in Table 2.1 on page 17. Note that nonterminals are italicized, keywords are bold-faced, and other reserved words are roman-typed. Square brackets ([,]) and vertical bar (|) are used for both as terminals and as nonterminals; those for terminals are underlined, e.g., [_;], and | are terminal brackets and a terminal vertical bar.

Type Specifications

\[
\text{typeSpec} \quad ::= \quad \text{type typeId} \\
\quad \quad \text{[parameters} \quad \{\text{paramList} \quad \rightarrow \quad \text{paramDecl} \} \\
\quad \quad \quad \quad \quad \rightarrow \quad \text{paramDecl}\}\} \\
\quad \quad \text{[mutation} \quad \text{mutationFlag]} \\
\quad \quad \text{supertypes} \quad \text{type}^+, \\
\quad \quad \text{usedTrait} \quad \text{traitId} \quad (\text{tToSMapping}) \\
\quad \quad \text{[invariant} \quad \text{invariant]} \\
\quad \quad \text{methodSpec}^* \\
\text{paramList} \quad ::= \quad \text{paramDecl}^+, \\
\text{paramDecl} \quad ::= \quad \text{paramId} \quad [< \text{type}] \\
\text{type} \quad ::= \quad \text{simpleType} \quad | \quad \text{paramType} \\
\text{simpleType} \quad ::= \quad \text{typeId} \\
\text{paramType} \quad ::= \quad \text{typeId}(\text{type}^+, \rightarrow \text{type}) \quad | \quad \text{typeId}(\rightarrow \text{type}) \\
\text{tToSMapping} \quad ::= \quad \{\text{type} \quad \text{for} \quad \text{sortId}\}^+, \\
\text{invariant} \quad ::= \quad \text{true} \quad | \quad \text{forall} \quad (\text{varId}: \text{type}) \quad [\text{assertion}] \\
\text{mutationFlag} \quad ::= \quad \text{true} \quad | \quad \text{false}
\]
```
typeId ::= identifier
paramId ::= identifier
traitId ::= identifier
sortBy ::= identifier

Method Specifications

methodSpec ::= methodHeader methodBody
methodHeader ::= msgPattern [returns formalDecl]
msgPattern ::= unary | binary | keyword
methodBody ::= [preCond] [modiList] postCond
unary ::= msgId
binary ::= binarySelector formalDecl
keyword ::= {msgId : formalDecl}+
formalDecl ::= formalId ::= type
binarySelector ::= opChar [opChar]
preCond ::= requires assertion
postCond ::= ensures assertion
modiList ::= modifies nothing
           ::= modifies at most selfOrFormalId+,
selfOrFormalId ::= self | formalId
msgId ::= identifier
formalId ::= identifier

Assertion Specifications

assertion ::= true | false | not(assertion) | (assertion)
            ::= assertion connectives assertion
            ::= quantifier (varDeclList) [assertion]
            ::= term = term | term \text{\&=} term

term ::= varId | selfOrFormalId | (term)
       ::= opId[(term+,)] | term infixOp term
       ::= selfOrFormalId.pre | selfOrFormalId.post
       ::= selfOrFormalId.obj | new(formalId+,

79
```
connectives ::= & | | => | <=
quantifier ::= forall | exist
varDecList ::= \{varId+, : sortId\}+
selfOrFormalId ::= self | formalId
infixOp ::= opChar [opChar]
formalId ::= identifier
opId ::= identifier
sortId ::= identifier
varId ::= identifier
identifier ::= letter [letterOrDigit*]
letterOrDigit ::= letter | digit
letter ::= A | B | ... | Z | a | b | ... | z
digit ::= 0 | 1 | ... | 9
opChar ::= + | - | * | / | \ | ~ | < | > | = | @ | # | $ |
         | ` | & | ? | ^ | ! | % | \
APPENDIX C

This appendix lists and gives a short explanation for each yellow button menu command in each subwindow of a system type browser and a system trait browser except for the text subwindow (see Chapter 3 of [29] for yellow button menus of text pane). A yellow button menu is a pop-up menu that allows a user to choose one of several actions to be performed on the contents of the selected window. It is opened by pressing the middle button (sometimes called yellow button) of the three-buttoned mouse (see Chapter 3 of [29]).

C.1 System Type Browser

Fig.C.23 shows the yellow button menu commands of each of the subwindows of a system type browser.

C.1.1 Type categories pane

file out  Creates a file in a standard format (similar to that of Smalltalk) containing definitions of all the types in the selected category. The file will have a default name typeName.lst and can be subsequently read back into the system. See Section 3.1.6.1 and Section 3.1.6.3.

print out Creates a file in PostScript format containing definitions of all the types in the selected category. The default file name is categoryName.ps. This file cannot be subsequently read back into the system. See Section 3.1.6.2.
Figure C.23: Yellow button menus of a system type browser

**spawn** Opens a *type category browser* on the selected category. A type category browser has the same functionality as a system type browser except that it limits access to the types in the specified category (see Fig. 3.13 on page 59). See Section 3.1.3.1.

**add category** Adds a new category to the system either before the selected category or at the end of the list if no category is currently selected. A prompt window will request the name of category to be added.

**rename** Changes the name of the currently selected category. A prompt window will request a new name.

**remove** Removes the currently selected category and any types in that category from the system. If any types are to be deleted, a *confirmer*\(^\text{17}\) will appear to request confirmation.

**update** Updates the information displayed in a browser. Changes to the type library made external to a browser (e.g., filing-in) is not automatically visible to the browser.

\(^{17}\)A *confirmer* is a window used to ask ‘yes’ or ‘no’ type of answers to a user. It is most often used to ask the user to confirm whether or not a request for some undoable action should be carried out. See Chapter 3 of [29].
edit all Displays the categories together with the types in each category in the text pane. The list may be edited to change the categories or the order in which categories are displayed. Changes must be accepted into the system by choosing accept entry from the yellow button menu.

find type Locates a specified type in a browser. A pattern may be specified using ‘*’ as a wild character in the prompter window asking the name of type to be located. If pattern is specified, all types matching the pattern, if any, are displayed in a list menu. Selecting one causes the browser to position itself at that type. See Section 3.1.1.1.

C.1.2 Type names pane

file out Creates a file in the standard format containing the definition of the currently selected type so that it can be subsequently read back into the system. The default file name is typeName.lst. See Section 3.1.6.1 and Section 3.1.6.3.

print out Creates a file in PostScript format containing definitions of the currently selected type. The default file name is typeName.ps. See Section 3.1.6.2.

spawn Opens a type browser on the currently selected type. A type browser limits access to only one type (see Fig. 3.13 on page 59 for a type browser). See Section 3.1.3.1.

spawn trait Opens a trait browser on the used trait of the currently selected type. A trait browser is a degenerated system trait browser (see Fig. 3.18 on page 66 for a trait browser). See Section 3.1.4 and Section 3.2.

definition Display the definition of the selected type in the text pane. The definition may be edited and accepted to the system using the yellow button menu of the text pane. See Section 3.1.1.2.

hierarchy Display the supertype/subtype hierarchy of the selected type. See Section 3.1.1.3.

protocols Display the entire message protocol associated with the currently selected type in the text pane. Depending on the instance-meta switch setting, display either instance or meta protocol. It may be edited and accepted into the system. See Section 3.1.1.4.

find method spec A list of all the messages specified by the currently selected type will be displayed, allowing the user to select the method specification to be viewed. See Section 3.1.1.6.
comment Display the comment associated with the currently selected type in the text pane. The comment may be edited and accepted to the system using the yellow button menu of the text pane.

rename Changes the name of the currently selected type. A prompt window will appear requesting a new name.

remove Remove the currently selected type from the system. A confirm will appear to request confirmation.

C.1.3 Message categories pane

file out Creates a file in the standard format containing all method specifications under the selected message category so that it can be subsequently read back into the system. The default file name is typeName-MessageCategoryName.lst. See Section 3.1.6.1.

print out Creates a file in PostScript format containing all method specifications under the selected message category. The default file name is typeName-MessageCategoryName.ps. See Section 3.1.6.2 and Section 3.1.6.3.

spawn Opens a message category browser on the currently selected type (see Fig. 3.13 on page 59). See Section 3.1.3.1.

spawn trait Opens a trait browser on the used trait of the currently selected type. A trait browser is a degenerated system trait browser (see Fig. 3.18 on page 66 for a trait browser). See Section 3.2 and Section 3.1.4.

add protocol Add a new message category to the selected type. A prompter will appear requesting the name of new category to be added.

rename Changes the selected message category name. A prompt window will appear requesting a new name.

remove Remove the selected message category name and any method specifications under that category. A confirm will appear to request confirmation if any method specifications are to be deleted.

C.1.4 Message selectors pane

file out Creates a file in the standard format containing the selected method specification so that it can be subsequently read back into the system. The default file name is typeName-MessageCategoryName.lst. See Section 3.1.6.1 and Section 3.1.6.3.
**print out** Creates a file in PostScript format containing the selected method specification. The default file name is `typeName-MessageCategoryName.ps`. See Section 3.1.6.2.

**spawn** Opens a *message browser* on the currently selected method specification (see Fig. 3.14 on page 60). See Section 3.1.3.1.

**spawn trait** Opens a *trait browser* on the used trait of the currently selected type. A trait browser is a degenerated system trait browser (see Fig. 3.18 on page 66 for a trait browser). See Section 3.1.4 and Section 3.2.

**specifiers** Open a browser which allows to browse all method specifications with the same selector as the currently selected one in the message selector pane (see Fig. 3.16 on page 62). See Section 3.1.3.3.

**implementors** Open a browser which allows to browse all Smalltalk methods that implement the currently selected method specification (see Fig. 3.16 on page 62). See Section 3.1.3.3.

**remove** Remove the selected method specification from the system. A confirmor will appear to request confirmation.

### C.1.5 Implementing classes pane

**spawn class browser** Opens a *class browser* on the selected implementing class. A class browser allow one to browse a Smalltalk class in the similar way as in a type browser (see Section 9.3 of [12] and Section 4.2 of [29]).

**spawn class hierarchy** Opens a *class hierarchy browser* on the selected class. A class hierarchy browser is similar to a type hierarchy browser except that it browses through the Smalltalk classes and the subclass hierarchy (see Section 9.3 of [12] and Section 4.2 of [29]).

**class ref** Opens a message-set browser on all the methods in the Smalltalk system that refer to the selected implementing class (see Section 9.3 of [12] and Section 4.2 of [29]).

**add implementing class** Adds a Smalltalk class to the set of implementing classes of the type currently selected in the type names pane. A prompt window appears asking to type in a class name.

**remove implementing class** Removes the selected class from the the set of implementing classes of the currently selected type in the type names pane.
C.1.6 Direct subtypes (supertypes) pane

**select** Makes the selected direct subtype (supertype) to be the selected type in the type names pane and its category to be the selected one in the type categories pane.

**spawn** Opens a type browser on the selected direct subtype (supertype) (see Fig. 3.13 on page 59 for a type browser).

**add direct subtype (supertype)** Makes a type to be a direct subtype (supertype) of the type currently selected in the type names pane. A prompt window appears requesting to type in the name of a type to be added.

**remove direct subtype (supertype)** Removes the selected subtype (supertype) from the set of direct subtypes (supertypes) of the type currently selected in the type names pane.

C.2 System Trait Browser

Fig.C.24 shows the typical yellow button menu commands of each of the sub-windows of a system trait browser.
C.2.1 Trait categories pane

**file out** Creates a file in a standard format (similar to that of Smalltalk) containing definitions of all the traits in the selected category. The default file name is `traitName.lsl`. The files created by **file out** can be subsequently read back into the system (see Section 3.1.6.3 for filing-in).

**print out** Creates a PostScript file containing definitions of all the traits in the selected category. The default file name is `categoryName.ps`. This file cannot read back into the system.

**spawn** Opens a trait category browser on the selected category. A trait category browser has the same functionality as a system trait browser except that it limits access to only the traits in the specified category (see Fig. 3.18 on page 66).

**add category** Adds a new category to the system either before the selected category or at the end of the list if no category is currently selected. A prompt window will request the name of category to be added.

**rename** Changes the name of the currently selected category. A prompt window will request a new name.

**remove** Removes the currently selected category and any traits in that category from the system. If any traits are to be deleted, a confirmor will appear to request confirmation.

**update** Updates the information displayed in a browser. Changes to the trait library made external to a browser (e.g., filing-in) is not automatically reflected by the browser.

**edit all** Displays the categories together with the traits in each category in the text pane. The list may be edited to change the category or the order in which categories are displayed. Changes must be accepted into the system by choosing **accept** entry from the yellow button menu of the text pane.

**find trait** Locates a specified trait in a browser. A pattern may be specified using "*" as a wild character in the prompter window asking the name of a trait to be located. If pattern is specified, all traits matching the pattern, if any, are displayed in a list menu. Selecting one causes the browser to position itself at that trait.
C.2.2 Trait names pane

file out Creates a file in the standard format containing the definition of the currently selected trait so that it can be subsequently read back into the system (see Section 3.1.6.3 for filling-in). The default file name is traitName.lsl.

print out Creates a file in PostScript format containing definitions of the currently selected trait. The default file name is traitName.ps.

spawn Opens a trait browser on the currently selected trait. A trait browser limits access to only one trait (see Fig. 3.18 on page 66).

add trait Display a trait template in the text pane so that a new trait may be added to the system by editing and accepting the template using the yellow button menu of the text pane.

definition Display the definition of the selected trait in the text pane. The definition may be edited and accepted to the system using the yellow button menu of the text pane.

comment Display the comment associated with the currently selected trait in the text pane. The comment may be edited and accepted to the system using the yellow button menu.

remove Remove the currently selected trait from the system. A confirmer will appear to request confirmation.

To rename a trait, bring the trait into the text pane by selecting its category and the trait in the categories pane and trait name pane respectively. Change the trait name in the text pane, accept it to create a new trait, and delete the old one by selecting it and choosing remove from the yellow button menu of the trait names pane.