1-24-1994

Gentle Introduction to Larch / Smalltalk Specification Browsers

Yoonsik Cheon
Iowa State University

Gary T. Leavens
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/36

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

TR94-01
Yoonsik Cheon and Gary T. Leavens

January 24, 1994

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011
A Gentle Introduction to Larch/Smalltalk Specification Browsers

Yoonsik Cheon and Gary T. Leavens

TR #94-01
January, 1994

Keywords: formal method, interface specification, Larch/Smalltalk, Smalltalk, specification browser.


© 1994 Yoonsik Cheon and Gary T. Leavens
Permission is granted for you to make copies of this document for educational and scholarly purposes, but not for direct commercial advantage.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
A Gentle Introduction to Larch/Smalltalk Specification

Browsers

Yoonsik Cheon and Gary T. Leavens*
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
cheon@cs.iastate.edu and leavens@cs.iastate.edu

January 24, 1994

Abstract

This paper provides a tutorial introduction to the Larch/Smalltalk specification browsers. The browsers are specification support tools providing a powerful and sophisticated environment for writing and managing Larch/Smalltalk specifications. They are integrated in the Smalltalk-80 programming system. The reader is assumed to have some familiarity with the Smalltalk system and Larch-style specification.

1 Introduction

For specifications to be practically and productively used in the programming, adequate support tools are needed. They help specifiers to check and maintain the consistency of formal text and assist in managing large volume of specifications. The Larch family of specification languages [GH93] provides a set of specification support tools such as syntax and sort checkers, and a theorem prover called LP [GH93, Chapter 7]. The Larch/Smalltalk specification browser provides a powerful and sophisticated environment for writing and managing Larch/Smalltalk specifications, which is integrated with the Smalltalk-80 programming system. Larch/Smalltalk [CL93a] is a Larch interface specification language for Smalltalk.

As writing Smalltalk programs is quite different from writing programs in the conventional programming languages, so writing Larch/Smalltalk specifications quite different from the traditional approaches to developing specifications. The major difference is the interactive and incremental style of specification development. By incremental, we mean that specifications are developed by adding some changes or additional features to the existing specifications. When writing a new specification, the specifier starts by inheriting properties from one or more existing specifications, modifies them, and extends them by specifying additional features. Thus specifications need not start from scratch. To start things off, there would be extensive on-line library of specifications, including specifications

*The work of both authors is supported in part by the National Science Foundation under grant number CCR-9108654.
for the Smalltalk library classes. Specifying a program module in terms of its difference from other program modules leads to shorter specification, and such specifications are easier to maintain. Writing specifications is done interactively. New or modified specifications are syntax- and sort-checked, and added to the library interactively using the powerful graphic user interface of the Smalltalk system. Thus Larch/Smalltalk specifications are not just plain text, but organized materials accessed and manipulated interactively through specification browsers.

The Smalltalk system provides a nice environment for integrating specifications into programming in the sense that specifications play a major role in the programming process. To this end, the Larch/Smalltalk specification browsers are developed on top of the Smalltalk system. Some of our design goals that lead us to the current specification browsers are:

- To integrate the specification browsers into the Smalltalk system in such a way that specifications can play a major role and be practically used in the programming process.

- To allow the specification browsers to be used as a front-end to the Smalltalk code browsers so that one can browse through Smalltalk programs in the subtype hierarchy (conceptual relationship), not in subclass hierarchy (implementation relationship).

- To have functionality and appearance similar to Smalltalk code browsers so that Smalltalk programmers can easily learn and use the specification browsers.

In the rest of this section we give a brief overview of Larch/Smalltalk specification language and specification browsers. In Section 2 we gives a tutorial introduction to the specification browser through a guided example. In Section 3 and 4 we describe other useful features of the specification browsers. Appendix A lists all the menu items available from the specification browsers accompanied by short descriptions.

1.1 Larch/Smalltalk

Object-oriented programming languages, such as Smalltalk [GR83], help one to build reusable program modules. However, the reuse of program modules requires adequate documentation — formal or informal [LC91, Coo92]. Larch/Smalltalk is a formal notation for specifying program modules in Smalltalk. It is a member of the Larch family of specification languages such as Larch/C [GH93, Chapter 5], Larch/Modula3 [GH93, Chapter 6], and Larch/C++ [LC92, CL93b]. In addition, it supports the notion of subtyping and inheritance of specification. The unit of specification is an abstract data type, which is implemented by one or more Smalltalk classes. The underlying abstract model of a type is specified equationally in the Larch Shared Language (LSL) [GH93, Chapter 4], and the behavior of its methods is modeled as a relation between program states and specified by Hoare-style pre- and post-conditions [Hoa69]. A type can be a subtype of other types, called its supertypes. A subtype can inherit specifications from its supertypes.

Figure 1 shows a specification for the method remove: of type Set(Elem). A method specification defines a message that can be successfully sent to the objects of the specified

\footnote{The current version of the Larch/Smalltalk specification browsers are implemented on the ParcPlace Objectworks\textsuperscript{TM} for Smalltalk, Release 4.}
remove: e <: Elem
  returns r <: Elem
  "remove e from the receiver and answer e."
  requires include(self.pre,e)
  modifies at most self
  ensures self.post = delete(self.pre,e) ∧ r = e

Figure 1: A specification of remove: method of type Set(Elem)

type, the type Set(Elem). All the method specifications together describe the protocol of the specified type. A method specification consists of the header and the body. The header gives information necessary to invoke the specified method while the body describes the effect of the specified method invocation, i.e., the effect of the corresponding message send. The header is similar to that of Smalltalk methods except that we decorate the input arguments with their types and optionally specify the result object and its type. If the returns clause is omitted, the receiver (self) is assumed to be returned by default. The body consists of a requires clause, an optional modifies clause, and an ensures clause. A requires clause specifies the pre-condition that must hold to invoke the specified method. If the pre-condition is not satisfied and the method is invoked, nothing is guaranteed. An omitted requires clause is interpreted as equivalent to "requires true"; i.e., the method can be invoked in any state. 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. An optional modifies clause lists those objects that may mutate their values as the result of the method evaluation. This is a strong indirect assertion that no other objects except for those listed are allowed to change their values. The default is "modifies nothing". The pre-condition constrains the clients while the modifies clause and the post-condition constrain the implementors.

This semantics of method specifications is a total correctness semantics; that is, if a method is invoked in a state in which the pre-condition holds, then the method evaluation must terminate and in that terminal state the post-condition must holds.

The specification of Figure 1 says that the method remove: takes an object of type Elem and if it is an element of the receiver (otherwise nothing is guaranteed), may mutate the receiver to make its final value equal to that of deleting the argument from its initial value. The reserved word "self" denotes the receiver, i.e., an object of Set(Elem) to which the specified message is sent. The notation "self.pre" means the value of self in the pre-state, the state just before method invocation, and "self.post" denotes the value of self in the post-state, the state just after method evaluation. The trait functions "delete" and "include" in pre- and post-conditions come from the trait SetTrait (see Figure 3), the abstract model for the type Set(Elem).

The connection between interface specifications such as the method remove: and the underlying abstract model (the trait SetTrait) is made when a type is specified; in fact, a method specification is a part of a type specification. A type specification consists of a header and a set of method specifications. For example, Figure 2 shows the header part of
The parameterized type Set(Elem). This parameterized type specifies a collection of related types. The type parameter Elem can be replaced by any types; thereby creating such new types as Set(Integer), Set(Character), Set(String), and so on. The direct supertypes of the specified type are listed after the keyword supertypes. For each type Elem, Set(Elem) is a subtype of Collection(Elem). A type inherits its supertypes’ properties including their method specifications. After the keyword usedTrait is specified the name of used trait which gives the underlying abstract model for the specified type. The used trait name is followed by a type-to-sort mapping in parentheses, which identifies the abstract values for each type in the specification. For example, the used trait of type Set(Elem) is the trait SetTrait (see Figure 3) and the type-to-sort mapping says that the type Set(Elem) is based on the sort S and the type Elem is based on the sort E. Thus, the abstract values of Set(Elem) are the equivalence classes of the terms of sort S: empty, insert(empty, e1), insert(insert(empty, e1), e2), etc., where e1, and e2 are terms of sort E.

Figure 3 shows the used trait SetTrait specified in LSL. It is an equational specification with some additional constructs such as generated by clause and partitioned by clause. An LSL trait denotes a theory in typed first-order logic with equality. A theory contains the trait’s assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. LSL provides a set of reusable traits in the form of LSL Handbook [GH93, Appendix A].

1.2 Basics of Larch/Smalltalk Specification Browsers

The Larch/Smalltalk specification browsers are opened from the ParcPlace Objectworks™ for Smalltalk system’s main menu, called the Launcher. Select the entry Larch/Smalltalk in the Launcher, then choose type browser on its pop-up menu to open a system type browser or choose trait browser to open a system trait browser. The system type browser is for viewing, entering, modifying, removing, and sort-checking Larch/Smalltalk specifications, whereas the system trait browser is for browsing through LSL traits. Figure 4 shows both the system type browser and the system trait browser.

Using the terminology of the Orange book [GoI84] and the more recent [LP90], a short description of the specification browsers is given here so that we can refer to them in the later sections. The look and functionality of the specification browsers are similar to the Smalltalk system class browsers [LP90, Chapter 4]. The system type browser (see Figure 4) is divided into eight scrollable panes (or views) and two switch panes labeled instance and meta. The top four and bottom three are list panes, while the center one is a text pane. Related type specifications are grouped into type categories, and related method specifications within an individual type are classified into message categories. The four
SetTrait(E,S): trait
  includes Integer
  introduces
    empty: → S
    insert, delete: S, E → S
    include: S, E → Bool
    isEmpty: S → Bool
    size: S → Int
  asserts
    S generated by empty, insert
    S partitioned by isEmpty, include
  for all s: S, e, e1: E
    delete(empty,e) == empty
    delete(insert(s,e),e1) == if e = e1 then delete(s,e1) else insert(delete(s,e1),e)
    include(insert(s,e),e1) == if e = e1 then true else include(s,e)
    size(empty) == 0
    size(insert(s,e)) == if include(s,e) then size(s) else size(e) + 1
    isEmpty(s) == size(s) = 0

Figure 3: A trait SetTrait

Figure 4: Larch/Smalltalk specification browsers; a system type browser (right) and a system trait browser (left). Also shown is a Smalltalk code browser showing an implementation of the currently browsed method specification.
upper panes from left to right list: type categories, type names, message categories, and method selectors. The switch panes below the messages category pane determines whether instance methods or meta methods are displayed in the message categories and method selectors panes; these are on-off switches, i.e., selecting instance deselects meta and vice versa. In the Figure 4 the system type browser lists four type categories (in the upper left pane): Kernel-Objects, Magnitude, Collections, and Graph-Examples, from which Graph-Example is selected. The type names displayed in the type names pane (second from the upper left) are therefore the types in the category Graph-Examples. The type Graph is selected and this, together with the fact that the instance switch is selected, means that both “accessing” and “testing” (displayed in the third upper pane from the left) are instance method categories of type Graph. The message category “accessing” is selected, implying that instance method specifications in that category of type Graph are displayed in the method selectors pane (in the upper right pane). Finally, the method selector addNode: is selected, thus specification for this method is displayed in the text pane.

The bottom three list panes show names of: implementing classes, direct subtypes, and direct supertypes of the currently selected type. In Figure 4, the specification Graph is implemented by class Graph as shown in the bottom left pane. For each type Node, Graph(Node) has two direct subtypes DirectedGraph(Node) and UndirectedGraph(Node), shown in the bottom middle pane. It also has one direct supertype Object, shown in the bottom right pane.

The system trait browser (Figure 4 on the left) consists of three panes; trait categories, trait names, and text. Like type specifications, traits are also grouped into categories, which are listed in the categories pane at the top left. All the traits in the selected category are listed in the trait names pane at the top. The text pane (below the others) shows the definition of the trait selected in the trait names pane. In the figure, for example, the trait category Graph-Example is selected, thus all the traits (DirectedGraph, Edge, Graph, Pair, and UndirectedGraph) in that category are shown in the trait names pane. Since the trait Graph is selected in the trait names pane, its definition is displayed in the text pane.

Specification browsers can be closed, collapsed, moved, and framed using the <window> button menu. A <window> button menu is the menu of actions that can be performed on any Smalltalk windows, such as close and move. It is brought up by pressing the right button (also called the blue button) of a three-buttoned mouse. When a browser window is deactivated or collapsed, the current selections are remembered and restored when the browser is reactivated or framed at some later time. Refer to Chapter 4 of [L90] for details on the Smalltalk user interfaces.

Appendix A gives short explanations all the <operate> menu items of each subwindow’s of the specifications browser. An <operate> 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 pane (see Chapter 3 of [L90]). It is opened by pressing the middle button (also called the yellow button) of a three-buttoned mouse.

2 A Guided Example

In this section, we develop a simple Larch/Smalltalk specification to show how to use the Larch/Smalltalk specification browsers. We will specify a parameterized type Set.
Writing a formal specification consists of many iterations of interacting steps. In Larch, there are constant interactions between specifying the interface components and writing the shared components (LSL traits). As interface specifications are developed, one might get deeper understanding of the problem, thus want to explore a different underlying mathematical models (e.g., using sequences instead of sets). A new method specification may require one to introduce new LSL sorts or trait functions, leading to refinement of the traits. The refined traits, in turn, may lead to respecification of some interface components. The iterative nature of developing formal specification is well supported by the Larch/Smalltalk specification browsers — support for graphical user interfaces and interactive and incremental style of writing specifications. Writing Larch specifications can be broken down into several steps [Win87]:

1. **Understand the problem.** An analysis of the problem is often best done by first identifying important concepts of the problems that can be represented by sorts (types), then the important relationships between them.

2. **Decide on the major abstractions.** For the interface component, write the header parts (type headers and method headers) and for the shared component, write syntactic information such as sort names and trait functions with their signatures.

3. **Fill in the blanks.** Write the body of method specifications (pre- and post-conditions). For traits, fill in the semantic information (axioms).

4. **Check.** Check whether the current specification adequately captures the intended meaning and repeat previous steps until they converge.

As specifications are developed, it is also necessary to evaluate them for certain properties such as consistency and completeness. The Larch Prover [GH93, Chapter 7] may be used to prove some of such properties. If one is documenting existing code, the steps will differ.

### 2.1 Specifying with Browsers

The above steps are realized with the specification browsers as follows: (1) create type to be specified and its used trait, (2) identify methods to be specified, (3) for each method, write its header part, (4) write the underlying mathematical model (the used trait), and (5) for each method, write its pre- and post-conditions.

**Step 1. Create type and used trait**

Larch/Smalltalk specifications are categorized in the system. This helps to organize the large number of types and methods that make up a complicate application. Type specifications are stored in categories. Similarly, a type’s method specifications are classified into categories. Traits are also categorized. Typically a new category is created for each application. This makes it easy for the application to be stored in a separate external file (see Section 3.1).

Thus, to create the type Set, we first need to add a new category, say, Example-Specs. Figure 5 shows this step. First decide where you want it to appear in the type categories.
To insert it above another category, select that category; to append it at the bottom of the list, make sure no category is selected. Select **add category** in the `<operate>` menu, which is opened by pressing the middle button of the mouse. The system will prompt for a new category name to be added. Enter the name `Example-Specs` (or blank to cancel the operation), then press `<return>`. The system creates a new category and it is selected. Also the type template appears in the text pane as shown in the top browser of Figure 6.

To enter a new type `Set`, we need to edit this template. Using the keyboard and the `<operate>` menu of the text pane, edit the template to replace dummy names with actual names. (Refer to Section 3.4 of [LP90] for details of editing text.) For example, `typeName` should be replaced by `Set`. `Set` will be parameterized with a type variable `Elem` which denotes the type of elements. And it will be a direct subtype of type `Object`. The used trait will be `SetTrait`, which will be defined later. So the specification should look like the one in the bottom browser of Figure 6.

When satisfied with the definition, select **accept** in the text pane’s `<operate>` menu (Figure 6). The system will check for syntax and other errors. If none is detected, the name `Set` will be displayed in the type names pane, in the proper alphabetic location; the type specification is created and stored in the system. In our case, the system will show an error message right before the name `SetTrait` saying “unknown trait”. This is because there is no trait named `SetTrait`. We did not create it yet. Remove the error message by pressing the `<delete>` key or by selecting **cut** in the `<operate>` menu.

To create the trait `SetTrait`, open a system trait browser from the Smalltalk Launcher (if one is not already open). Like types, traits are also stored in categories. Create a new category `Example-Specs` in a similar way as for type categories, then a template will appear in the text pane (see Figure 7).
Figure 6: Adding a new type Set. The top browser shows the type template and the bottom browser shows the definition of type Set as it is accepted.
Change the dummy #TraitName to #SetTrait and select accept from the text pane’s <operate> menu (see the bottom browser of Figure 7). Then the name SetTrait will appear in the trait names pane and a trait template in the text pane (see the top browser of Figure 8). Edit the template to produce the definition shown in the bottom browser of Figure 8 and select accept from the text pane’s <operate> menu to store it in the system. This is just a stub needed to accept the type Set; it will be elaborated at the later steps. Ad the sorts S and E appear in the type-to-sort mapping of the type definition, we define two trait functions “empty” and “insert” with their signatures.

Now come back to the system type browser and accept the Set specification. This time it should be installed in the system without any error messages.

**Step 2. Identify methods to be specified**

In this step we will decide what methods the type Set will have. There are two kinds of methods in Larch/Smalltalk: *instance methods* and *meta methods*. Instance methods define messages that can be sent to an instance (object) of the specified type. Meta methods describe Smalltalk’s class messages. A method specification is classified as instance or meta when it is entered into the system. The following methods will be specified for the type Set. All are instance methods except for the method new.

- instance creation: new
- add/remove: insert:, remove:
- accessing: size, choose
- testing: isEmpty, include:
Figure 8: Specifying trait SetTrait by editing the trait template

Since method specifications are stored in categories, the method categories are also identified in this step. For example, in the category "add/remove" we have two methods **insert:** and **remove:** The intention of each method is given by its descriptive method selector name. The details of each method will be specified in further steps. At this point we have not yet told the Larch/Smalltalk browser about their method names (and categories).

**Step. 3. Write the interface of each method**

At this step we specify the interface of each method identified in the previous step. By interface we mean the input and output arguments and their types. Using the **modifies** clause, we may also want to specify that what input arguments are allowed to be modified.

As mentioned before, method specifications are classified into categories. So, let us first add into the browser all the message categories identified in the previous step. Adding a new message category is done similar way as was done for a type category. For example, to add a new category “add/remove”, first set the instance/meta switch properly; i.e., in this case, select the **instance** button in the instance/meta switch pane. Next select the **add protocol** from the <operator> menu of the message categories pane. Then a prompter window will appear asking for a message category name to be added. Type “add/remove” and **accept**. If no message category was selected when **add protocol** was invoked, the new category will be added at the end of the list of categories. Otherwise, it will be inserted before the selected category in the list. Repeat this to add all the message categories and
make sure to select meta button in the instance/meta switch pane when adding the message category “instance creation”.

To add a new method specification, first select the message category that applies to the method or if a message category is already selected, deselect it. For example, to add an instance method specification for the method remove:, first select the instance button in the instance/meta switch pane and select the message category “add/remove”. Then a method specification template will be displayed in the text pane (see the top browser of Figure 9). To specify the remove: method, replace the dummy names in the template with the actual names. For example, the specification of method remove should look like the one in the bottom browser of Figure 9.

The specification says that the method remove: takes an object of type Elem and returns the receiver itself. An omitted returns clause means that by default the receiver will be returned after the method evaluation. The method can mutate the receiver. Nothing is said about the behavior of the method (pre- and post-condition) at this point. A comment
in the specification can be enclosed inside a pair of double quotes though omitted in the above example.

Once done with editing, for readability and syntax check, select **format** in the *<operate>* menu of the text pane. The message selectors and keywords will be bold-faced. To sort-check and store the method specification into the system, select **accept** in the *<operate>* menu (see Figure 9). If no error is detected, the specification will be installed in the system. Otherwise, a highlighted error message will be inserted at the point where the error is detected. If that is the case, using the *<operate>* menu remove the error message and edit the text again (see Section 2.2).

Repeat the above step for each method to specify its arguments and their types; that is, write header information for methods `new`, `insert`, `size`, `choose`, and `isEmpty`. Remember to set the instance/meta switches to meta when the meta method `new` is specified (see Figure 10).

**Step 4. Specify the Underlying Mathematical Model**

Now we need to write the shared component (the trait `SetTrait`) to specify the underlying mathematical model\(^2\). The trait provides us vocabulary (sort names and trait function names) to specify pre- and post-conditions of method specifications.

A major concern in constructing traits is deciding what axioms are needed. The notion of sufficient-completeness is a helpful technique for writing traits [Gut77, GH78, LG86]. In a trait there is a *distinguished sort*, sometimes called the *type of interest* or *data sort*. For

---

\(^2\)In reality one could use the trait from the LSL Handbook ([GH93, Appendix A]).
Figure 11: A system trait browser showing the complete specification of trait SetTrait.

For example, the sort S in the SetTrait is the distinguished sort. The distinguished sort's trait functions can be categorized as *generators*, *observers*, and *extensions*. A set of generators produces all the values of the distinguished sort. The observers are the operators whose domain includes the distinguished sort and whose range is some other sort. The extensions are convenience functions that can be defined using the generators and observers. A good heuristic for writing enough equations to adequately define an abstract type is to write an equation defining the result of applying each observer or extension to each generator. That is, write equational axioms for each terms resulting from applying an observer or an extension to each combination of generators. Refer to [LG86] for more details on this topic.

Figure 11 shows a complete specification of the trait SetTrait. The generators are "empty" and "insert", and observers are "include", "size", and "isEmpty". The extension is "delete". The trait was created by first editing previous definition and then accepting it using the <operate> menu of the text pane.

**Step 5. Write pre- and post-conditions**

For each method specification, write its pre- and post-conditions. To do this, first select its message category in the message categories pane, and choose its method selector in the method selectors pane. The current specification will appear in the text pane. For example, to bring up current definition of method remove:, choose "add/remove" in the message category pane, and then select remove: in the method selectors pane. Now one will see in the text pane the definition one entered before. Write the pre- and post-conditions using the <operate> menu of the text pane.

For example, Figure 12 shows the specification for the remove: method. The pre-
condition says that the method **remove**: can be invoked only with an object of **Elem** that is an element of the receiver. The post-condition asserts that the argument will not be an element of the receiver anymore. Since the **returns** clause is omitted, the specification implicitly states that the receiver (**self**) is returned by default. To store the new definition in the system, select **accept** from the `<operate>` menu of the text pane (see Figure 12). If there is no error, the system will store it as new version; otherwise, an error message will be inserted at the point where the error is detected (see Section 2.2).

Repeat the above step for each method specification to complete its pre- and post-conditions. Figure 13 shows the complete specifications of all the methods in the textual representation. Additional LSL trait functions might be needed to write assertions of method specifications. If that is the case, edit the used trait and store it by accepting through the system trait browser. Note that all the LSL trait functions appearing in the pre- and post-conditions must be defined in the system before the method is accepted; otherwise, the system will complain about them.

### 2.2 Fixing Errors in Specifications

When a specification is stored in the system by selecting **accept** in the `<operate>` menu of the text pane, it is first syntax and **short-checked**. Assertions in the pre- and post-conditions should sort-check in the sense that LSL trait function applications conform to their signatures. Larch/Smalltalk sort inference rules can be found in [CL93a]. Visual feedback is provided when such errors occur by inserting an error message at the point in the specification where the error was discovered. The message is highlighted so that it can be easily noticed and deleted.
**type** Set
  
  **parameters** Elem
  
  **supertypes** Object
  
  **mutation** true
  
  **usedTrait** Set Trait(Set(Elem) for S, Elem for E, Integer for int)

**new**
  
  **returns** s :< Set(Elem)
  
  **ensures** s = empty ∧ fresh(s)

**insert**: e :< Elem
  
  **modifies at most** self
  
  **ensures** self.post = insert(self.pre, e)

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

**choose**
  
  **returns** e :< Elem
  
  **requires** not(isEmpty(self.pre))
  
  **ensures** include(self.pre, e)

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

**include**: e :< Elem
  
  **returns** b :< Boolean
  
  **ensures** b = include(self.pre, e)

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

Figure 13: A Larch/Smalltalk specification of parameterized type Set
As a simple exercise, let us create a simple error. Bring the specification of method `remove:` to the text pane of the system type browser by making proper selections. Its pre-condition is: “include(self.pre.e),” That is, the element to be removed (e) must be an element of the receiver (self.pre). Change “include(self.pre.e)” to “include(e,e)” and select `accept` from the `<operate>` menu of the text pane to store the new specification. Then one will get an error message saying that the system cannot sort-check the new pre-condition “include(e,e),” This is shown in Figure 14. The error message is inserted just before the symbol `e` which caused the error. It says that `e`'s sort is `E` while the trait function “include” expects a term of sort `S`; remember that the signature of the trait function “include” is “S, E -> Bool” (see the trait SetTrait). To correct the error, remove the error message by pressing the `<delete>` key (or selecting `cut` from the `<operate>` menu of the text pane), change `e` to `self.pre`, and `accept` it again.

3 Saving Specifications

When a new Smalltalk system image is created, any changes made to Larch/Smalltalk specifications are automatically saved into the new image. A new snapshot of the Smalltalk system image can be created either at any time during a session by selecting `save` from the Launcher, or when quitting from Smalltalk by making proper selection in the `quit` sub-menu of the Launcher (see Chapter 4 of [LP90]). However, a typical way of saving specifications is to store them to external files so that they can be subsequently read back into the system. This processes are known as `filing-out` and `filing-in` in Smalltalk jargon. Filing out specifications is a good way to share specifications with others, to archive them, and to save changes.

3.1 Filing Out/In

Larch/Smalltalk specifications can be filed out at four different levels: (1) a single method specification, (2) all the method specifications of a particular message category, (3) a type specification (including all its method specifications), and (4) all the type specifications in a selected type category. Each list pane in the system type browser provides a file out
menu entry corresponding to the four levels of output. When one selects file out, one is prompted for a file name. The default is the name of the method, type, or category with the extension “.lst”, which stands for Larch/Smalltalk. For example, Figure 15 shows filing-out of all the types under the category “Example-Specs” to a file Example-Spec.lst.

The system trait browsers have a similar facility to file out a trait or a collection of traits in a selected category. The default file name extension is “.lst”.

A text file containing Larch/Smalltalk or LSL specification as formatted by the file out can be subsequently read back into the system. The File List is used to file in specification exactly the same way as one does with Smalltalk code (see Chapter 4 of [LP90]); that is, select the file in the list pane of a File List, then choose file in from the code pane’s <operate> menu. An alternative way is to open a File Editor on the file containing specifications, highlight the text and then select file it in from the <operate> menu. Yet another alternative is to evaluate:

\[
\text{FileStream fileNamed: } \text{fileName} \text{ fileIn}
\]

where fileName is the desired Larch/Smalltalk or LSL specification file name written as a string in single quotes. As the specification is filed in, the System Transcript displays the names of the file and, when applicable, the name of each type (or trait) and message category.

Changes external to specification browsers are not automatically reflected by the browsers so one must choose update on the <operate> menu of type (or trait) category pane to reflect newly filed-in specifications.

---

A File List is a special Smalltalk system browser used to list the contents of a directory or file, edit a file, and create a new file (see Chapter 4 of [LP90]). It is opened from the Launcher.
3.2 Printing

The menu entry `print out` also appears in each of the list pane of the system type browser (and the system trait browser). When selected, it writes the corresponding specification to an external file in a pretty-printed format suitable for reading by a human reader. The current version generates a PostScript™ file. Files generated using `print out` can not be subsequently read back into the system using `file in`.

4 Other Features

4.1 Browsing through Type Hierarchy

An interesting feature of Larch/Smalltalk is that even though the Smalltalk programming language does not support types and subtyping, one still can think about them in specifications. This in turn gives us the freedom to use a behavioral notion of subtyping for Larch/Smalltalk. That is, each object of a subtype should behave like some object of each of its supertypes [Lea91, Ame91]. Larch/Smalltalk type specifications are organized according to this subtype relationship. Behavioral subtyping is not enforced but is more useful for organizing specification than other notions. Larch/Smalltalk also supports inheritance of specifications; a subtype can inherit specifications from its supertypes, e.g., instance method specifications and invariants. This allows a new specification to be composed by stating how it differs from existing ones. Specifying a type in terms of its difference from supertypes leads to shorter specifications, and such specifications are easier to maintain. Thus, a type specification should be viewed in the type hierarchy induced by this subtype relationship. As in Smalltalk, at the top is the type `Object` which specifies properties that are inherited by all objects. All other types are direct or indirect subtypes of `type Object`.

The bottom-center pane of the system type browser lists the names of the currently selected type's direct subtypes if any (see Figure 4 on page 5). And the bottom-right most pane lists the names of the currently selected type's direct supertypes. For example, in Figure 4, type `Graph(Node)` has two direct subtypes `DirectedGraph(Node)` and `UndirectedGraph(Node)` respectively, and one direct supertype `Object`. Using the panes' `<operate>` menu (see Figure 20 on page 25), one can manipulate the set of direct supertypes/subtypes of the currently selected type. To browse a particular direct supertype/subtype, first select it from the corresponding pane and click the `select` item from the `<operate>` menu. Then the browser jumps to that type, i.e., it becomes the currently selected type of the browser. Using this feature one can easily navigate through the type hierarchy.

Sometimes it is useful to browse all the types along the supertype chains (up to the type `Object`), for example to view the full protocol supported by a type. A `type hierarchy browser` expedites this viewing of types in the supertype chains (see Figure 16). It is opened by choosing `spawn hierarchy` entry from the `<operate>` menu of type category pane. The type hierarchy browser is organized around type hierarchies rather than around type categories. However, in structure and functionality, it is similar to the system type browser, except that the types displayed are limited only to the supertypes of the selected type. Figure 16 shows a type hierarchy browser opened on the type `DirectedGraph`, a direct subtype of type `Graph` which is a direct subtype of type `Object`. 
4.2 Browsing Traits

In the previous example, we used the system trait browser to view and edit the used trait of the currently selected type. The system type browser provides another way to access it. The <operate> menus of the type names pane, the message categories pane, and the method selectors pane all contain the entry spawn trait, which opens a trait browser on the used trait. A trait browser is a specialized system trait browser which limits access to only one trait. It has the same functionality as that of a system trait browser.

4.3 Browsing Implementations

An implementation of a Larch/Smalltalk type is a set of Smalltalk classes that collectively conform to the specification. For example, we can say that Smalltalk classes Boolean, True, and False together are a legal implementation of the Larch/Smalltalk type Boolean; however, any of these classes in isolation does not have enough implementation to conform to the specification of type Boolean. The ability to browse implementations directly from the type browser has several important consequences. First, it is possible to browse through Smalltalk classes by their conceptual relationships (subtyping) rather than by their implementation relationships (subclassing). That is, the type browser can be used as a front-end to the Smalltalk system class browser. This is particularly useful when one browses through the large number of library classes. Second, one can develop a specification and its implementation simultaneously. Our experience suggests that specifying a program module and implementing it affect each other, thus they should be done hand-in-hand. Finally, while browsing through Larch/Smalltalk specifications one can always looks at the actual implementation (whether it is a class or a method) on the fly for implementation details.

The left-most pane of the bottom three panes of the system type browser lists the names of Smalltalk classes which implement the currently selected type (see Figure 20 on page 25).
Figure 17: Browsing implementations (classes and methods) from the system type browser

Using its <operate> menu, one adds or deletes a class to the set of implementing classes. To browse a particular implementing class, first select its class name in the pane and click the <operate> menu. Then a Smalltalk class browser will be opened on the selected implementing class (see Figure 17).

An implementation can also be browsed at the method level. As in types, a legal implementation of a Larch/Smalltalk method is a set of Smalltalk methods that collectively conform to the specification. For example, the method ifTrue:ifFalse: of type Boolean is implemented by three Smalltalk methods, all named ifTrue:ifFalse:, from three Smalltalk classes Boolean, True, and False. One can view all the Smalltalk methods that collectively implement the currently selected method specification. This ability to view the implementations (methods in Smalltalk) for a particular method specification (i.e., method level specifications-to-implementation link) is particularly useful in simultaneously specifying and implementing Smalltalk programs. To browse the implementations of a method, say addNode:, first bring its specification to the text pane by making proper selections in the message category pane and in the method selector pane. Next select the implementors entry from the <operate> menu of the method selector pane. Then, a message-set browser (Smalltalk code browser) is created on the implementing methods for the specification addNode: (see Figure 17).
4.4 Specialized Browsers

In the previous subsections we introduced several specialized browsers such as the type-hierarchy browser, the trait browser, and the message-set browser. They are specialized in the sense that they provide more limited views or views that are not organized along type/trait category boundaries, whereas the system type/trait browser provides access to the entire specifications in the system. It is often convenient to create browsers that provide this limited views, for example (1) to browse specific type category, type, message category, or method specification, (2) to browse a type in a specific supertype or specification inheritance chain, (3) to browse a set of related method specifications.

Each list pane of a browser such as the system type browser has a `spawn` entry in its `<operate>` menu to create corresponding specialized browser. A `type category browser` (see Figure 16) is opened by selecting `spawn` from the `<operate>` menu of type category pane. A type category browser provides access only to the types in a specified category. Except for this, it has the same functionality as a system type browser. A `type browser` allows access to only a specified type (see Figure 14 on page 17). In all other aspects, it has the same functionality as a type category browser. A type browser is created by choosing the `spawn` from the `<operate>` menu of type names pane. A `message category browser` and a `message browser` are two other specialized browsers (see Figure 18). 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. They are are created in the similar way by choosing the `spawn` entry from the `<operate>` menu of message category pane and method selector pane respectively.

`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 the same selector or 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 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 `<operate>` menu of the method selector pane after the method specification to be examined is chosen.
As in the system type browser, the system trait browser provides two specialized browsers: *trait category browser* and *trait browser* (see Figure 19). A trait category browser has the same functionality as a system trait browser except that it limits access to only traits in a selected category. A trait browser can browse only one trait.

5 Summary

Through a guided example we have shown how to use Larch/Smalltalk specification browsers to write specifications interactively and incrementally. The specification browsers also allow one to browse Smalltalk classes according to their conceptual relationships (i.e., subtyping). We expect that specification support tools such as the Larch/Smalltalk browsers help specifications to be practically used in the programming process.

The current version of Larch/Smalltalk specification browsers is implemented on the ParcPlace Objectworks™ for Smalltalk, Release 4 and is available by anonymous ftp from ftp.cs.iastate.edu under the directory /pub/larchSmalltalk.
References


[CL93a] Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. Technical Report TR #93-24, Department of Computer Science; Iowa State University, Ames, IA, October 1993. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.


Figure 20: <operate> menus of a system type browser


A Appendix

This appendix gives short explanations for the <operate> menus (sometimes called yellow button menu) of panes of the system type browser and the system trait browser. Since the menu items for the text pane are the same as that of Smalltalk class browsers, they are omitted here (see Chapter 3 of [LP90]). An <operate> 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 pane. It is opened by pressing the middle button (sometimes called yellow button) of the three-buttoned mouse.

The <operate> menu items for the specialized browsers such as type category browsers and type browsers are the same as the system type browsers.

A.1 System Type Browser

Figure 20 shows the <operate> menu commands of each pane of a system type browser.
A.1.1 Type Categories Pane

file out Creates a file in a standard format containing definitions of all the types in the selected category. The file will have a default name CategoryName.lst and can be subsequently read back into the system. See Section 3.1.

print out Creates a PostScriptTM file containing definitions of all the types in the selected category. The default file name is CategoryName.ps.

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.

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. The system prompts for a category name to be added.

rename Prompts for a new category name, then replace the currently selected category name with the new one.

remove Deletes the currently selected category and any types in that category. If any types are to be removed, a confirmert 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) are not automatically reflected to an opened browser.

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 <operate> menu.

find type Locates a given 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.

A.1.2 Type Names Pane

file out Store the selected types and its method specifications to an external file in a form that can be subsequently read back into the system. The default file name is TypeName.lst (see Section 3.1).

print out Creates a PostScriptTM file containing specification of the currently selected type. The default file name is TypeName.ps.

A confirmert 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 [LP90].
**spawn** Opens a *type browser* on the currently selected type. A type browser limits access to only one type. Except for that it has the same functionality as a type category browser.

**spawn trait** Opens a *trait browser* on the used trait of the currently selected type. A trait browser is a degenerated system trait browser.

**definition** Display the definition of the selected type in the text window. The definition may be edited and accepted to the system using the *<operate>* menu of the text window.

**hierarchy** Display the supertype/subtype hierarchy of the selected type.

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

**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.

**comment** Display the comment associated with the currently selected type in the text window. The comment may be edited and accepted to the system using the *<operate>* menu of the text window.

**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 confrmer will appear to request confirmation.

### A.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*.

**print out** Creates a PostScript™ file containing all the method specifications in the selected message category. The default file name is *TypeName-MessageCategoryName.ps*.

**spawn** Opens a *message category browser* to browse all the method specifications under the currently selected message category.

**spawn trait** Opens a *trait browser* on the used trait of the currently selected type. A trait browser is a degenerated system trait browser.

**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 confirmer will appear to request confirmation if any method specifications are to be deleted.

A.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 $\text{TypeName-MethodSelectorName}.lst$.

print out Creates a PostScript\textsuperscript{TM} file containing the selected method specification. The default file name is $\text{TypeName-MethodSelectorName}.ps$.

spawn Opens a method browser on the currently selected method specification.

spawn trait Opens a trait browser on the used trait of the currently selected type.

specifiers Open a browser which allows to browse all method specifications with the same selector as the currently selected one in the method selector pane.

implementors Open a Smalltalk code browser which allows to browse all Smalltalk methods that implement the currently selected method specification.

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

A.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 [Gol84, LP90].

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.

class ref Opens a message-set browser on all the methods in the Smalltalk system that refer to the selected implementing class.

add implementing class Adds a Smalltalk class to the set of implementing classes of the currently selected type. A prompt window appears asking a class name to be added.

remove implementing class Removes the selected class from the the set of implementing classes of the currently selected type in the type names pane.
Figure 21: <operate> menus of a system trait browser

A.1.6 Direct Subtypes/Supertypes Pane

select  Makes the selected direct subtype/supertype to be selected in the type names pane and its category to be selected in the type categories pane.

spawn  Opens a type browser on the selected direct subtype/supertype.

add direct subtype/supertype  Enter a type as a direct subtype/supertype of the currently selected type. A prompt window appears requesting to type in the name of a type to be added.

remove direct subtype/supertype  Deletes the selected subtype/supertype from the set of direct subtypes/supertypes of the currently selected type in the type names pane.

A.2 System Trait Browser

Figure 21 shows typical <operate> menu commands of each pane of a system trait browser.

A.2.1 Trait Categories Pane

file out  Creates a file in a standard format containing definitions of all the traits in the selected category. The default file name is CategoryName.lsl. The files created by file out can be subsequently read back into the system.

print out  Creates a PostScript™ file containing definitions of all the traits in the selected category. The default file name is CategoryName.ps.

spawn  Opens a trait category browser on the selected category. A trait category browser has the same functionality as a system trait browser. However it limits access to only traits in the specified category.
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. The system will prompt for a new category name to be added.

rename Changes the name of the currently selected category.

remove Removes the currently selected category and any traits in that category from the system. If any traits are to be deleted, a confirmer 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 all 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 <operate> menu of the text pane.

find trait Locates a given 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 the trait names matching the pattern, if any, are displayed in a list menu. Selecting one causes the browser to position itself at that trait.

A.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. The default file name is TraitName.lsl.

print out Creates a PostScript™ file 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 is a specialized system trait browser which limits access to only one trait; it has the same functionality as that of a system trait browser.

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 <operate> 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 <operate> 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.

rename Changes the name of selected trait. The system will prompt for a new name.

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