VISTA - A VISUAL INTERFACE FOR SOFTWARE REUSE IN TROMLAB ENVIRONMENT

Rajee Nagarajan

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 1999

0-612-43556-3

Abstract

VISTA - A Visual Interface for Software Reuse in TROMLAB Environment

Rajee Nagarajan

Software reuse has the potential to increase productivity and reduce development costs. Several research and experience reports show that practicing and achieving high levels of software reuse in the presence of repository and object-oriented computer-aided software engineering (CASE) development methods pose technical and managerial challenges. This thesis presents reuse research that seeks to integrate and enhance software development activity in TROMLAB framework, an environment being constructed at Concordia University for a rigorous development of real-time reactive systems. We propose a browser-based reuse framework for automatic search and retrieval of TROMLAB components. VISTA is a visual medium embedding the browser for practicing software reuse in TROMLAB applications. The browser provides facilities to navigate the repositories of reusable components, inspect components and their dependencies, and enable the retrieval and modifications of components for reuse. The browser also works in conjunction with the graphical user interface of TROMLAB, thus providing an environment for active reuse.

Acknowledgments

I am very much indebted and grateful forever to my mentor, Professor V. S. Alagar, for his invaluable supervision, patience, support and advise throughout the course of my studies at Concordia University. Dr. Alagar's technical, ethical and moral sensitivity have inspired me. I found working with Dr. Alagar tremendously rewarding and truly feel that I have learned from the best! I sincerely hope that our research relationship continues in the future.

I am grateful to Dr. Peter Grogono for his academic advise and encouragement throughout my undergraduate degree at Concordia. I am thankful to Dr. Kasi Periasamy for his encouragement and discussions on formal specification languages.

Although too numerous to mention, I would like to thank Dr. Gregory Butler, Dr. Jamil Hasan and Dr. T. Radhakrishnan for their continous encouragement. I am thankful to Dr. Terrill Fancott and Dr. Sheridan K. Houghten for their refreshing viewpoints and suggestions.

Special thanks to Stan Swiercz and all the system analysts in the Department of Computer Science for their instant technical support. I am grateful to Robby Kirk and Instructional & Informational Technology Services (IITS) for their warmth and friendship. I thank the *ROSE* research group members for their valuable discussions about **TROMLAB** environment.

Most heartfelt thanks goes to Dr. Debra J. Richardson, University of California (UC), Irvine, for her enriching constant encouragement and support and for providing me the excellent research facilties and environment to complete this thesis.

Last, but not the least, *ROSATEA* research group members in UC, Irvine, deserve my gratitude for their warmth and earnest friendship.

Contents

List of Figures

Chapter 1

Introduction

Reactive systems are characterized by their continuous ongoing stimulus-response relationship with their environment. For real-time reactive systems, the stimulus-response behavior is regulated by timing constraints. Such systems find application in areas such as transportation, process control, telephony, and strategic defence systems. In general, a reactive system has *infinite* behavior: a process in a reactive system is usually non-terminating. In this respect reactive systems are different from common transformational systems which may be regarded as functions from inputs available at the start of the computation to outputs provided on termination. Reactive systems are also different from interactive systems, such as human-computer interface. The major distinction is in the available synchronization mechanism: an interactive system waits for a reply from its environment; on the contrary a reactive system, such as an alarm system in a boiler plant controller, is fully responsible for synchronization with its environment. Consequently, a reactive system must satisfy two important requirements:

- **stimulus synchronization:** the process is always able to react to stimulus from the environment;
- **response synchronization:** the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment so that the environment is still receptive to the response.

Designing a real-time reactive system is an extremely difficult and challenging task. The complexity arises from three sources: (1) the large number of requirements, some of which may become available or well-understood only at later stages of design; (2) a large number of different executions of the system due to the orderings and interleavings of concurrent events; and (3) the real-time constraints on the event occurrences and the execution sequences. These difficulties and the increasing importance of real-time reactive systems in safety-critical applications necessitate continued research into methodologies and tools for specifying and reasoning about the design of such systems. One of the ways of reducing the complexity barrier is to develop components that are dependable, and reuse these components in a large-scale development environment. This thesis addresses issues involved in reusing reactive components.

During the last ten years software and hardware engineers have started employing graphical representations, such as state machines, timing diagrams, and data flow graphs to describe the properties of software systems that they design. Such visual representations capture and communicate the designer's intuitive understanding of a system. Many tools based on visual formalisms are now commercially available. Most notable ones are STATEMATE [17], ObjecTime [38], and Rational Rose [34, 35]. There is a formal semantics for STATEMATE; however, the other two graphical notations do not have formal semantics. Although research in cognitive sciences have revealed [20] that graphical notations do indeed facilitate human comprehension, only notations supported by a formal semantics provide well-defined meaning and promote logical reasoning. TROMLAB [2, 3] is an object-oriented development environment founded on (TROM) *Timed Reactive Object Model* formalism [1]. It is being designed and developed at Concordia University for the design and development of real-time reactive systems. The building blocks for developing reactive systems come from Larch and TROM formalisms. These basic components have a formal semantics. The objective of this thesis has been to develop a tool, called VISTA, supporting the maintenance and management of reusable reactive components in TROMLAB environment.

Recent research has shown that software development methodologies that emphasize reuse are increasingly recognized in industries in terms of the increased productivity and reduced software costs. However, in order for the process of software development to improve significantly there must be investment in tools that promote software reuse. One of the key ingredients for promoting software reuse is to build software repositories, and develop methods for retrieving components from the repositories. In the absence of such a tool to support developers in their search for identifying and retrieving software components, it is not possible to achieve a satisfactory level of reuse wherein its benefits are fully recognized. The capabilities of VISTA are appropriate for inspecting, and reusing repository components through visual interaction.

Viewing the development of software as an information management problem is not new; however, the methods that we use to design different repositories, one for each kind of component, and relate them according to TROM development methodology are new and novel within the context of reactive system development. There exists several browsers, the internet supports several. Browsers have been developed for developing Z [27] and Larch [29] specifications. However, the tool that we have developed is both more general and specific. The browser developed in this thesis is more general than Z and Larch browsers in the sense that in addition to developing Larch specifications, it can be used in a much wider context of TROMLAB. It is specific because it coexists and cooperates with other tools in TROMLAB environment. The browser allows navigation through components stored at different layers of abstractions, which include *Larch Shared Language* (LSL) traits, Larch/C++ (LCPP) interface specifications, TROMs, subsystems, and their relationships. The tool also manages different versions of each component, and inheritance (and subtype) relationships among components. The tool provides a specification development environment for LSL traits, invokes Larch syntax checker and helps the user develop syntactically correct LSL traits.

The browser can be invoked either independently or in conjunction with a Graphical User Interface (GUI), another tool in TROMLAB environment. Together they provide a comprehensive visual framework for accessing both static and dynamic information of the system. Invoking the browser at any instant from GUI, one can view and search among the static information in the databases as well as the dynamic information generated up and until the invocation of the browser. Thus, the tool combines two approaches:

- The provision of a combined visibility of the development process and the evolving product.

- The integration of product development process under reuse.

The tool also permits users to deal with two principal working aspects in an integrated fashion:

- The definition, enactment, and visualization of the development process.
- The animation of heterogeneous prototypes constructed by reusing the components in the reuse library.

The browser can be used for the following phases:

- **Process Model Support** -TROMLAB environment integrates formal approaches with the different stages of a process model. Accordingly, a formal model of the software unit is constructed by putting together a formalized set of requirements with a formal model of the environment. This software unit is then simulated to validate the requirements. Iterative design, simulated debugging and reasoning, and formal verification are the successive stages of the process model. The browser gives visibility and control over the entire set of components used for the development. This includes querying the database and retrieving suitable components for reuse in the design, the customization of LSL specifications, the monitoring and management of prototypes of the system under development.
- **System Model Configuration** - A system is constructed by putting together components, which may be objects instantiated from TROM classes, or other subsystems. Users can inspect system components and modify system configurations selected from the reuse library.
- **Design Animation** - Animation is the central piece of the simulator [30] which is used to validate system requirements. Developers can view the repository for versions of simulation histories and choose a previously simulated scenario for reasoning.

Some of the components in the reuse library, such as LSL traits, and TROMs may be useful in other contexts as well. For example, LSL traits may be reused in developing Larch interface specifications for reusable class libraries. TROMs are extended finite state machines, and may be useful to model the behavior of hybrid systems when real-time computations are permitted in the states. Consequently, VISTA may find applications outside of TROMLAB environment. Keeping this in mind, we have designed and implemented the tool so that it may be adapted to reuse activity in any application domain where LSL and TROMs are basic components. The rationale behind our design is to make the browser easy to use by both novice and experienced users. The following are the main contributions of this thesis:

1. A rationale for *black-box* reuse of TROMLAB components.

2. Design and development of VISTA, a visual tool for inspection, navigation, adaptation, and informal reasoning integrated with version management of components used in TROMLAB environment.

3. An interactive facility for LSL trait specification development environment. Users can browse and reuse LSL traits from a library in composing their specifications.

4. Link to Larch/C++ interface specification repository which can be used by the implementation phase of TROMLAB environment, when it becomes available.

Chapter 2

Technical Issues for an Effective Management of Software Reuse

The concept of software reuse was introduced in a seminal paper by McIlroy[26] at the 1968 Nato Software Conference—the founding date of Software Engineering discipline. He stated "I would like to see the study of software components become a dignified branch of Software Engineering. I would like to see standard catalogs of routines classified by precision, robustness, time-space requirements and binding time of parameters". Although some progress has been reported [6, 11, 19], it is the case that twenty five years later, software reuse has not yet become a widely accepted practice for software construction. Both technical and managerial issues must be integrated in promoting software reuse.

Recognizing the most obvious benefits of software reuse, such as productivity gain and improved software quality, many commercial, government and industrial organizations in the USA, Japan, and Europe have recently instituted systematic software reuse programs [23]. From the published reports of their experiences, we can infer that there is no consensus on what approaches work best; it is only clear that no single approach to software reuse has been found to be best for all applications. There are numerous diverse technical and non-technical issues challenging the wider practice of software reuse. Among them, the most important technical issue is the language used for creating, labelling and retrieving software components for error-free reuse. The choice of language has a profound impact on both technical and manegerial aspects of reuse. These include (a) supporting technologies and libraries used to accomplish reuse; (b) economic conditions on which reuse practice is justified; and (c) development, maintenance, and management of all future reusable assets such as requirements, specification, design, and high-level archtectures. The language chosen for documentation and implementation of reusable products is also closely related to an effective management of reuse program. In order to maximize the return on the investment of the reuse program, an organization should choose the language acceptable to a large community of users and adaptable for large-scale reuse. Based on the language, strict guidelines and standards for developing software components, or for acquiring software libraries to be used in the development process can be formulated. Moreover, the choice of language impacts the maintenance and management of reuse spectrum from domain models to domain specific frameworks and architectures. The language for composing specifications in TROMLAB has a simple syntax. The graphical user interface provides a means to compose diagrammatic representations. Consequently, the reuse factor is likely to be high in TROMLAB environment.

In our view, to fully realize the potential of reuse, software components for fault-free reuse must be made available with three pieces of information: *abstraction*—for unambiguous and precise understanding; *implementation* for code reuse; and *environment* —for enrichment and enhancement. Abstraction of the component is a precise unambiguous description of its functionality. Implementation provides the details of concrete data structures and algorithms for implementing the abstraction in a computer. Environment is the contextual information in which the software component is designed to perform. The language chosen for reuse must support abstraction, implementation, and contextual constraints.

Given that TROM formalism includes Larch specifications, and C++ is widely used in industries, this thesis advocates Larch/C++ language for specifying the interface specifications - thus, linking designs to C++ implementations.

2.1 Why Reuse?

Software development is becoming increasingly capital-intensive, tool dependent, co-operative, and requires greater early investment of capital in return for reduced cost at later stages. Thus reusable sofware is to be viewed as a capital good whose investment cost is recoverable amortized over a number of years from its large pool of users. The additional expected benefits of reuse are:

- It can lead to improved reliability and performance.
- Reuse promotes interoperability between systems.
- It supports rapid prototyping.
- It provides a transparent uniformity among systems constructed with the same components, which in turn leads to sound maintenance.

Recent reports from industries such as Hewlett-Packard[23], NEC and Fujitsu[6, 11], Digital[14], and IBM[13] also suggest that there is a reduction in defective software, increase in return on investment, improved efficiency in product delivery and in general, a corporate-wide acceptance of reuse. Hence it can be argued that the systematic application of reuse to prototyping, development, and maintenance during the entire software development process is an effective way to reduce cost and improve software reliability and productivity.

2.2 Scope of Reuse

Systematic software reuse is a paradigm shift in software development from building monolithic systems to families of related systems. This kind of paradigm shift was successfully applied in the 70's for developing families of algorithms based upon the design principles of divide and conquer, backtracking etc. The potential effect of the shifting paradigm in software reuse is expected to be profound when error-free reusable software components can be repeatedly assembled in different applications by knowing only their interface specifications.

There are two perspectives to software reuse: one involves careful design for reuse and is known as forward engineering; the other approach is design recovery and reverse engineering where software engineers try to recover and adapt existing requirements, design and code for reuse in different applications. However, it is strongly believed that technologies for recovering reusable objects from legacy systems will face immense challenges. Moreover, it may even be the case that much of existing software has little or no reuse value due to its lack of specifications, poor design, undependable code and imprecise documentation. Reuse, being the use of existing software component in a *new* context, must necessarily be a *design* issue. That is, a system must be designed with reusability in mind; otherwise, components will be extremely hard to reuse even when coding is correct and the documentation is adequate. This may be accomplished through abstraction - in initial modeling, in data and functions, in interface design, and in process definitions. The following issues play a key role in the design of reusable components:

1. A component with reuse potential must have a precise and unambiguous description of its functionality. So, internally the function must be *complete*. Externally, its *coupling* (interface) to the environment must be simple and well defined.

2. A component for reuse must be governed by abstraction principles supporting generality so that a family of dissimilar funtions may be derived from one description.

3. The *interface* of a software component must be made simple and must be standardized. Only then, it will have a large set of potential users and the extent of reuse will be high.

4. Object-Oriented (OO) concepts such as encapsulation, inheritance, and subtyping should be used in developing reusable hierarchies.

5. Reusable software should be easy to modify and be easy to compose.

6. Precise documentation must be provided for software components.

When a software component is developed to satisfy the above principles and is shown to meet its requirements, it can be released for reuse. It is clear that both technical and managerial teams should collaborate to maintain and distribute reusable products developed according to the above principles.

2.3 Reuse Management

In spite of several challenges facing widespread reuse, some software development organizations are making good progress in establishing and following reuse programs.

In the US, commercial organizations like AT&T and DEC[14], HP[23], IBM[6], Microsoft, and Motorola[6] have instituted reuse programs and report improved productivity and software quality. In Japan, software reuse in different forms have been practiced institutionally for more than a decade. For example, at the NEC Software Engineering Laboratory a software reuse library for its business application was built and then integrated into their software development environment. This enforced reuse in all of their business software applications which resulted in a 6.7:1 productivity improvement and 2.8:1 quality improvement. It is also reported [6] that in Perth, Australia, the reuse program at the Universal Defense Systems has accumulated a library of 396 Ada modules and expects this to grow. In Europe, Siemens, the European Space Agency and ESPRIT have reported major efforts in establishing reuse programs. The following major conclusions may be derived from these reports: (1) a strong emphasis must be put on integrating reuse tools with software development environment; (2) measurements to show quality improvement and productivity improvement must be kept; (3) full participation in software reuse effort at all stages of software development must be encouraged.

2.3.1 Reuse Experiences

It is to be expected that because of the encapsulation and data abstraction provided by the OO paradigm, programmers could reuse classes by understanding the behavior of classes as specified by the methods in their interfaces. The client of a class may view the method implementations as being contained in a *black-box* hidden from view. That is, the black-box approach to reuse assumes that a classes's behavior can be described succintly and without having to refer to an implementation. In many manuals the description of reusable class libraries are given only in a natural language. These are often imprecise, verbose, and potentially ambiguous. Due to these shortcomings, class descriptions give rise to different interpretations among reusers, often forcing them to read and understand the implementation details. The effort required to understand the implementation details of a method is often much more than the effort required to rewrite it from scratch. All this wasted efforts offsets all or part of the productivity increases which should have resulted in effective reuse of the module. A remedy is to provide natural language descriptions accompanied by some formal descriptions.

In code reuse, it is clear that most of the simple classes such as container classes (lists, dictionaries, queues), strings, date, time, and tracing package would be required by all applications. This is the first level of reuse. In the second level, class designs that are more domain specific would be reused either individually or in clusters by more sophisticated developers. Informal descriptions of class designs and their interactions in this level are extremely hard to read and understand. As the reuse process matures more and more complex structures, such as design patterns and frameworks, would be reused. Their informal descriptions and internal details are much more difficult to comprehend, if not accompanied by precise descriptions. This thesis addresses issues related to the second level of reuse in the context of TROMLAB environment.

These factors clearly suggest that, in large part, reuse efforts would be thwarted by the absence of documentation specifying the behavior of reusable components in a precise and complete fashion. The absence of precise and complete interface documentation can also result in the mis-interpretation of the intended behavior of available components by developers. This may lead to system integration and system maintenance problems, warranting much time to be spent during system integration to remove the defects. If the behavior of components is not precisely defined, it is also difficult to determine whether these problems were due to incorrect usage of reusable components or due to the fact that the components were defective.

Black-box reuse has a number of advantages:

- Black-box reuse is simpler for the reuser. Much of the benefits of reuse is lost if developers are forced to understand the implementation details of a component in order to understand the behavior of the component.

- The real value of reusable code lies in its properties, such as correctness with respect to its formal specification. When changes are made in the code, the program's behavior is no more predictable and consequently there is no more invariance of properties. Hence, black-box reuse prevents developers from making assumptions of their own about the implementation.

- For polymorphic code, failure to rely solely on the specification makes it difficult or impossible to determine what constraints client subclasses must respect to ensure predictable and reliable behavior.

- Reuse products can be thoroughly tested and classified before release.

- An efficient maintenance of a reuse library is possible when the products are not subject to modifications.
- Managing a black-box reuse program is effective because standard conventions can be imposed for using the products.

2.3.2 Formal Specifications and Black-Box Reuse

Black-box reuse is easy to learn, practice, and administer. What kind of software is suitable for this level of reuse? If the software is not useful and does not perform as intended, then it does not matter how reusable it is; it is simply worthless as an asset. Only after a developer locates and understands the usefulness of the software its reuse can be planned. The first step in this planning is to determine the behavior of the software to see whether it will serve the purposes in the plan. Here is where a good documentation and description of the behavior and interfaces to the product becomes essential; module descriptions and programming code are simply insufficient for black-box reuse.

In our opinion, black-box reuse represents a higher level of reuse than just source code reuse since it is akin to design reuse. Developers reuse not only the implementation code of a component but also the abstract design of that component, as described in the component's interface specification. For well-designed, and well documented reusable software this is very significant, as a considerable amount of effort will have gone into the design of the behavioral interface. In the case of frameworks of classes, the amount of design reuse is even more substantial since the developer reuses not only the design of individual classes but also the design of the interactions between classes, which are notoriously difficult to design properly.

In order to truly support the ability to reuse a component without having to understand its internal details, the components's interface must convey the abstraction implemented by the class in a precise and complete manner. Trying to provide a precise interface specification using standard natural language specifications can be a very challenging and time consuming task. One of the reasons is that, unless great care is taken, providing the necessary level of precision results in an overly verbose specification which is difficult to read and understand, thereby reducing the effectiveness with which a component can be reused. Recognizing the necessity of precise interface specifications several researchers in object-oriented community[39, 21] have advocated the use of formal specifications to specify component interfaces. The syntactic support for creating reusable black-boxes are not sufficient; the semantic support required to state the behavior of components must also be provided. TROMLAB components have well-defined syntax and semantics. Consequently, black-box reuse is appropriate in TROMLAB environment.

When the behavior of a reusable component is only informally stated, the reuser will either misunderstand the component's behavior (resulting in improper use of the class) or turn to read the implemented code (resulting in both loss of time and rigor). To remedy this from happening, the functionality of a software component must be stated with precision and conciseness so that all users understand its behavior in one and only one way. A formal specification language due to its mathematical apparatus and formal semantics enable the developer of software to write the functionality unambiguously. In turn, the reuser can build correct systems from these descriptions.

There are a number of additional advantages to use formal specifications in the context of developing and using reusable software components. Organizing and retriving reusable classes from class libraries have only a limited scope when purely syntactic considerations are used. Organizing reusable classes according to the subtype hierarchy provides a logical and meaningful view of classes. Moreover, behavioral subtyping relations between classes cannot be deduced without a precise specification of the behavior of these classes. The proof system in the formal specification language can be used to formally verify the correctness of a component's implementation against its specifications.

2.4 Linking Designs to Implementations - Choice of Language

In TROMLAB environment there are three categories of components. The abstract data type layer is founded on Larch[15], and consequently Larch/C++ interface specifications link the abstractions to a C++ implementation. The Larch family of languages have the following advantages:

- Larch provides a two-tiered approach to specification. In one tier a Larch Interface Language (LIL) (C++ in the case of Larch/C++) is used to describe the semantics of C++ program modules. In the other tier, Larch Shared Language (LSL) is used to specify state-independent, mathematical abstractions, called *traits*, which can be referred to in any LIL specification. Hence, the two tiers provide a clean separation of concerns.

- The Larch/C++ specification of a C++ module specifies not only the behavior of the function, but also how exactly the function is called from C++ code.

- Externally observable behavior can be expressed independent of implementation details.

- Functions are described in Larch/C++ using Hoare-style pre- and post- conditions.

- The LSL tier is declarative and assertions can be stated and proved using LP, the Larch Theorem Prover.

- The syntax for data members and member functions in interface specifications are almost the same as in a C++ program.

Specifications written in Larch/C++ for implementable components serve a useful purpose in reusing components. The reuse repository includes a large number of LSL traits and Larch/C++ specifications developed in the context of formalizing the interface specifications for a number of classes chosen from the Rogue Wave Tools.h++[36], a rich, robust and versatile C++ foundation class library. Virtually any programming chore can be done using the classes from this library. Tools.h++ is an industry standard. The functionalities provided by Rogue Wave Tools.h++ library suits the requirements of a wide range of applications in large-scale industrial development of software. It is claimed[36] that the Rogue Wave Tools.h++ class library is built to achieve four basic goals: *efficiency*, *simplicity*, *compactness*, and *predictability*. This perfectly fits in with the criteria for black-box reuse.

Chapter 3

Requirements of the Browser

In this chapter, we identify the requirements to build VISTA with which developers in TROMLAB environment can interact to access, manage, and navigate the reuse repository of TROMLAB components. In our context, the term *component* is used to denote elements of the software that are predefined in terms of the general functionality that they can provide. The components may denote the software artifacts used within TROMLAB as well as its architectural design parts. The activities that affect them are the evolutionary changes that occur amidst interaction of TROMLAB components. The users of VISTA are the developers and other clients of TROMLAB environment who share reusable software components for building reactive systems. Below we give a brief overview of TROMLAB architecture and defer detailed structural and behavioral descriptions of the reusable components to the next chapter.

3.1 TROMLAB Environment

TROMLAB is an Object-Oriented (OO) software development environment for constructing complex *real-time* reactive systems built with TROM. TROMLAB environment supports specification, design, validation and verification of subsystems built with TROMs. The environment proposes [2, 3] a two-pronged strategy to contain complexity: the object-oriented framework for modeling reactive system supports iterative design, design refinements and hence reduces design complexity; the animator and the verification subsystems provide the tool support necessary for validating the design and verifying time-dependent properties during the evolution of design. Consequent to these design decisions, the components arising in design include TROMs and components included in them, components assembled using TROMs, and components arising from their refinements. The browser includes the reuse libraries of the components, the editor, and a query processor interacting through a graphical user interface of the TROMLAB.

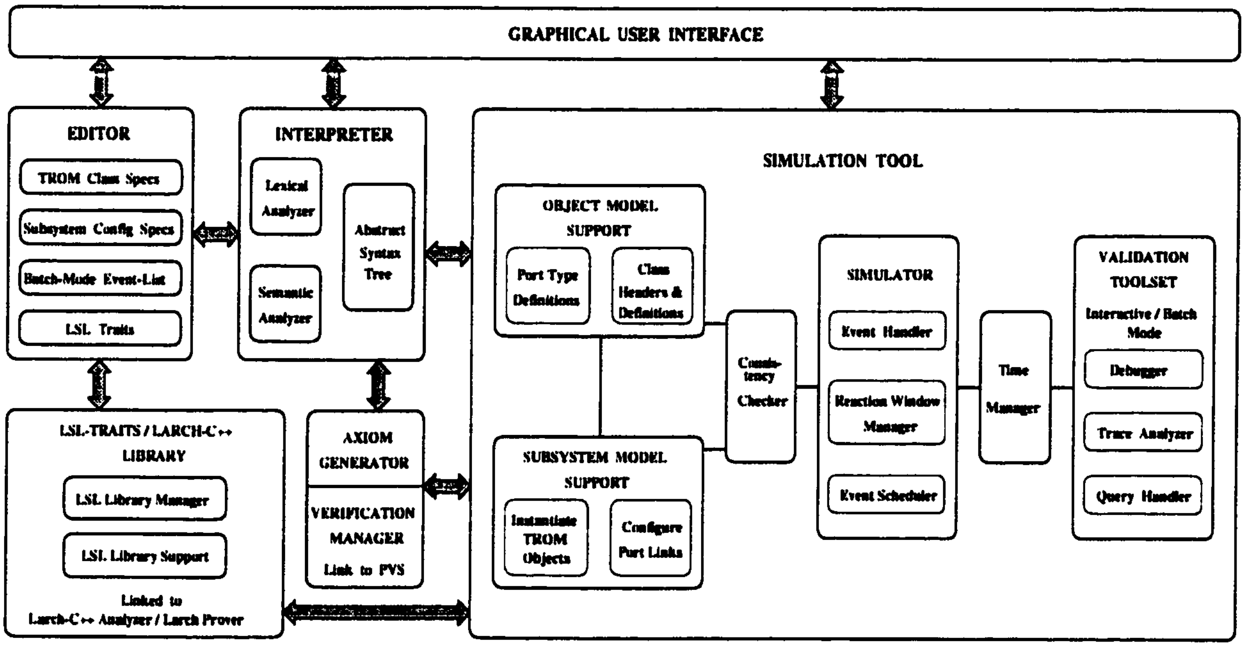


Figure 1: Structure of TROMLAB

Figure 1 shows the context and role of VISTA within the design components of TROMLAB.

- **LSL traits and Larch/C++ Library** - This is a passive component, a library of LSL and Larch/C++ specifications. The specifications in this repository were composed, analyzed, and used by researchers in the reuse research project at Concordia University [4]. All specifications in the repository have been checked for syntactic correctness and in particular, all Larch/C++ specifications conform to the requirements stated in Rogue Wave library [36]. All specifications developed by TROMLAB users can also be stored and retrieved from this library; however, they are maintained separately from the pre-tested components. The specifications developed by users of TROMLAB are submitted to Larch tools and checked for syntactic accuracy and analysed for semantic consistency.

- **Editor** - New LSL traits can be composed, and others may be modified using an editor facility. TROM class specifications, and subsystem specifications may also be textually composed; however, GUI facilities are better suited for this purpose.
- **Interpreter** - The TROM specification of a reactive object is presented as a *class* definition. The interpreter performs lexical and semantic analysis on the TROM class definitions and on the specification of subsystem configuration. The static analysis of TROM classes that are input to the simulator can be done by the interpreter via the editor. If the TROM is syntactically correct then it builds an internal representation (AST) of each TROM class and the *SCS*. At this point, specifications from the lowest tier (LSL traits) are compiled separately and checked for consistency using LP, Larch Prover. The TROM classes in the middle tier can import the LSL traits and are compiled separately independent of other tiers. New states, transition specifications or time constraints can be added incrementally to class specifications, and recompiled before inclusion to a subsystem. When the *SCS* in the upper tier is redefined, it can be separately recompiled without requiring a recompilation of components in other tiers.
- **Axiom Generator** - It uses the logical semantics of TROMs and the subsystem including them to generate a set of axioms for the specific subsystem. There are three types of axioms: **transition specifications**; **time constrained axioms**; and **synchrony axioms**. These axioms are used by the simulator for on-line reasoning about the system behavior. In the TROM class specification, the actual times of occurences of events, the number of ports, and the actual port at which an event occurs are dynamic information. They can be obtained only during the instantiation of object configuration and object collaboration. But the axioms generated with the static information of an object of a TROM class specification can be used by all the subsystems that includes this object. Such axiomatization of TROM ensures completeness of its logical behavior for each predicate in the axioms and for each state, event and port. The predicates are reducible to propositions at any instant by the simulation environment. Thus, reasoning of system properties becomes decidable in the simulation environment. The browser may interact with the animator to browse and import Larch/C++ interface specifications and C++ programs required for a reasoning session.

- **Simulation Tool** - This is to perform system validation through simulator. It's debugging facilities include freezing the simulation and activating the validation toolset. When simulation is frozen, it allows the user to interact with the process for injecting input events, and query the behavior of the system being simulated. The user can walk through the event trace and examine the history of the simulated scenario. This allows the user to roll back and analyze the change in the system status and simulate iteratively the corrected designs. Also, the safety properties at each each step in the design can be verified.

- **Graphical User Interface (GUI)** - A user interacts through the GUI to compose specifications, to submit them for syntactic and semantic analysis, to invoke the simulator, tu watch the dynamics of simulated system, and to interact with the browser at any stage of the development activity and inspect and navigate system components. Interpreter, simulation tool, and verification manager interact with the browser through GUI.

- **Verification Manager** - This component verifies time-dependent properties in TROM classes and subsystems using PVS [32].

- **Browser User Interface** - This provides the interface for Larch reuse repository when the browser is invoked independently from TROMLAB. It also interacts with GUI to enable the viewing, navigation, and retrieval of TROMs and subsystems.

The Interpreter and the Axiom Generator were designed and implemented by Tao [41]. The Simulation Tool was designed, and implemented by Muthiayen [30]. Some fundamental work leading to formal verification and the design of a Verification Manager can also be found in Muthiayen [30]. Based upon the works of Tao and Muthiayen, a reasoning system and simulated debugging facilities are being designed by Haidar [16]. The GUI is being designed by Srinivasan [40]. Upon completion of these two works, we expect a full prototype of TROMLAB to become operational.

3.2 Requirements

From the outline given in the previous section, we identify the requirements of VISTA, the browser tool. These are:

1. VISTA should enable users to read and view LSL traits and Larch/C++ specifications from the library.

2. VISTA should provide a facility for composing new Larch traits with or without reuse of LSL traits from the library.

3. VISTA should interact with Larch tools to check the syntactic correctness of composed LSL traits as well as analyze some of their properties.

4. It should be possible to maintain the relationships among LSL traits. These include the language specific relationships **includes**, and **assumes**, and design relationships, called **versions**.

5. It should be possible to run ail versions of LSL library traits as well as versions of composed traits against LSL syntax checker. Only syntactically correct traits must be saved. Users must be given an oppurtunity to modify and recompose specifications.

6. VISTA enables the maintenance of relationships between LSL traits and Larch/C++ interface specifications.

7. Relationships between Larch/C++ specifications must be maintained. This is the *inheritance* hierarchy defined in Rogue Wave Library [36].

8. The relationship between a Larch/C++ interface specification and an implementation (C++ program) of its member functions must be maintained.

9. VISTA should maintain a repository of syntactically correct TROMs. When requested by GUI, it should release or record TROM specifications from the repository.

10. VISTA should maintain the following relationships among TROMs:

    - A TROM should be related to all its refinements as defined by the refinement theory of TROMs [1].
    - A TROM should be related to all LSL traits included in its definition.
    - A TROM should be related to all subsystems in which an instance of it is included.

11. It should maintain a repository of all subsystems composed from syntactically correct TROMs and subsystems. When requested by GUI, it should release or record TROM specifications from the repository.

12. The following relationships among subsystems should be maintained:

    - A subsystem should be related to all TROMs instantiated in its definition.
    - A subsystem should be related to all other subsystems included in its configuration.
    - The port links connecting objects in a subsystem definition must be recorded.

13. VISTA should provide navigation and viewing facilities for the above components and their relationships. In particular, the following basic queries and suitable combinations of them must be answered:

    (a) Retrieve a trait based on name.

    (b) Retrieve a trait based on **includes** (**assumes**) relationship.

    (c) Retrieve a trait based on version number.

    (d) Retrieve a Larch/C++ specification based on name.

    (e) Retrieve a Larch/C++ specification based on **uses** relation.

    (f) Retrieve a Larch/C++ specification based on inheritance information.

    (g) Retrieve traits included in a given TROM specification.

    (h) Retrieve TROMs included in a given subsystem specification.

    (i) Retrieve ports and port links in a given subsystem configuration.

    (j) Retrieve all traits transitively related by **includes** (**assumes**).

    (k) Retrieve all Larch/C++ specifications transitively related through **uses** relation.

The browser should also be able to respond to queries composed from the above basic queries. Two such examples are:

1. Retrieve the versions of all TROMs included in a given subsystem.

2. Retrieve the transitive closure of all traits mentioned in the **uses** clause of a given Larch/C++ specification.

The browser can be run either in conjunction with or independent of GUI. These modes can be switched at any time. To run the browser successfully, the following are the software and hardware requirements:

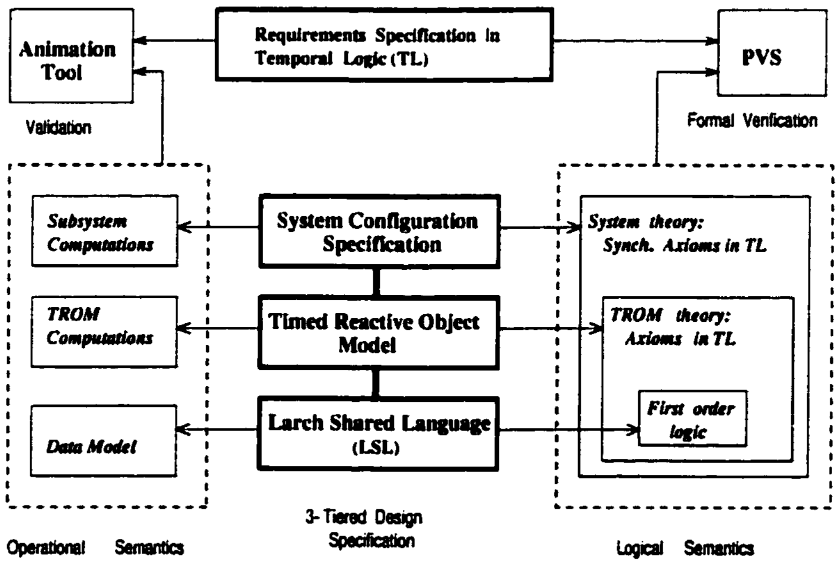
1. The Larch syntax checkers *lsl*, *lcpp*, and the Larch Prover *LP* must be available.
2. The browser should run on a Sun SPARC station 10 using Solaris 2.3 operating system.
3. It should interact with GUI, and through GUI with the rest of TROMLAB system.

Finally, the browser must remain useful for both experienced and naive users of TROMLAB and Larch environment. It should be adaptable to evolving need of the TROMLAB user community.

Chapter 4

Reusable Components of TROMLAB Environment

Having identified the functional requirements, software and hardware requirements, and performance requirements of the browser, we discuss in this chapter the structural and behavioral aspects of the reusable components. In TROMLAB there are three



Animation Tool
Requirements Specification in Temporal Logic (TL)
PVS
Validation
Formal Verification
Subsystem Computations
System Configuration Specification
System theory: Synch. Axioms in TL
TROM Computations
Timed Reactive Object Model
TROM theory: Axioms in TL
Data Model
Larch Shared Language (LSL)
First order logic
3-Tiered Design Specification
Operational Semantics
Logical Semantics

Figure 2: Overview of TROM Methodology

basic component types:

- LSL traits, and Larch/C++ interface specifications
- TROM, the generic reactive object model

- SCS, subsystem specification

These components are structurally and behaviorally different; however, TROM methodology, as we briefly review below, uses them at three distinct levels and uses components from one level for constructing components in the next higher level. Thus, when components from a lower level are included in composing a component in a higher level, the structure and behavior of included components affect the structure and behavior of the defined component.

TROM methodology is a fusion of Object-Oriented(OO) methodology with real-time technology. It is a three-tiered methodology [1] as shown in (Figure 2). The top most tier constitutes the *System Configuration Specification* (SCS) to describe the object interaction. The middle-tier specifies the detailed specification of reactive objects for a particular problem as described in the requirements. The lowest tier specifies the data abstractions in Larch Shared Language (LSL) used in TROM class definitions of the middle tier.

Since this methodology is OO, it inherits natural OO techniques such as *modularity, reuse, encapsulation*, and *hierarchical decomposition* using *inheritance*. This feature enhances the incremental development that allows the composition, verification and integration of large and complex systems. In order not to confuse with the notion of *objects* in OO, we have called the reusable elements of the three tiers as *components*. Below we give a description of the components belonging to these levels.

4.1 Larch Specifications

Larch is a property-oriented specification language. It uses a two-tiered approach to formal specifications. The first tier, using Larch Shared Language (LSL), provides programming language independent specifications defining the structure and behavior of abstract data types and general theories of objects. Each unit of LSL specification is called a *trait*. A trait specifies either a data type or any theory to be combined with a data type. The second tier, using Larch Interface Languages (LIL), provides specifications of the components of a software system. The interface language is particular to the programming language used for the software system and defines the interfaces of the components of the system. The traits in the LSL tier can be referred in the interface tier. An LSL trait is written in the equational algebraic style. Each equation is an axiom in first order predicate logic. The interface specifications are written using Hoare style pre- and postconditions.

```
SymTab : trait

    introduces
        emp :→ S
        add : S, K, V → S
        rem : S, K → S
        find : S, K → V
        isin : S, K → Bool

    asserts
        S generated by (emp, add)
        S partitioned by (find, isin)
        ∀ (s : S, k, k1 : K, v : V)
            rem(add(s, k, v).k1) == if k = k1 then s else add(rem(s, k1), k, v)
            find(add(s, k, v), k1) == if k = k1 then v else find(s, k1)
            isin(emp, k) == false
            isin(add(s, k, v), k1) == (k = k1) ∨ isin(s, k1)

    implies
        converts (rem, find, isin) exempting (rem(emp), find(emp))
endSymTab
```

Figure 3: LSL Trait Specification for a Symbol Table

A simple LSL tier Larch specification for a symbol table is shown in Figure 3. This specification introduces the trait *SymTab*, a theory describing a symbol table. The signature of *SymTab* introduces five functions in the **introduces** clause. The **asserts** clause introduces the equations constraining the function. The implies clause states the checkable redundancies, which logically follow from the equations. The **generated by** clause states that all symbol table values can be represented by terms solely composed of two function symbols *emp* and *add*. This clause defines an inductive rule of inference and is useful for proving properties about all symbol table values. The **partitioned by** clause adds more equivalence between terms i.e., two terms are equal if they cannot be distinguished by any of the functions listed in the clause. This can be used to prove that insertion is commutative. The **exempting** clause documents the absence of right sides of equations for *rem (emp)* and *find(emp)*. Error situations are handled in the interface specification. The **converts** and **exempting** clauses together forms the completeness of the symbol table specification. There are other clauses such as **includes**, **assumes**, and **implies** which may be used in composing a trait. See trait $SymTab_1$, which is a refinement of *SymTab*.

```
init = proc() returns (s:symbol_table)
            ensures $s' = emp \wedge new(s)$

insert = proc(s:symbol_table, k:key, v:val)
    {
            requires $\neg isin(s, k)$
            modifies $(s)$
            ensures $s' = add(s, k, v)$
    }

lookup = proc(s:symbol_table, k:key) returns (v:val)
    {
            requires $isin(s, k)$
            ensures $v' = find(s, k)$
    }

delete = proc(s:symbol_table, k:key)
    {
            requires $isin(s, k)$
            modifies $(s)$
            ensures $s' = rem(s, k)$
    }
end symbol_table
```

Figure 4: Larch Interface Specification for a Symbol Table

4.2 Structure of TROM

Larch/C++ interface specification language is appropriate as a language of implementation for components of TROMLAB. the specification in the next tier. Larch/C++

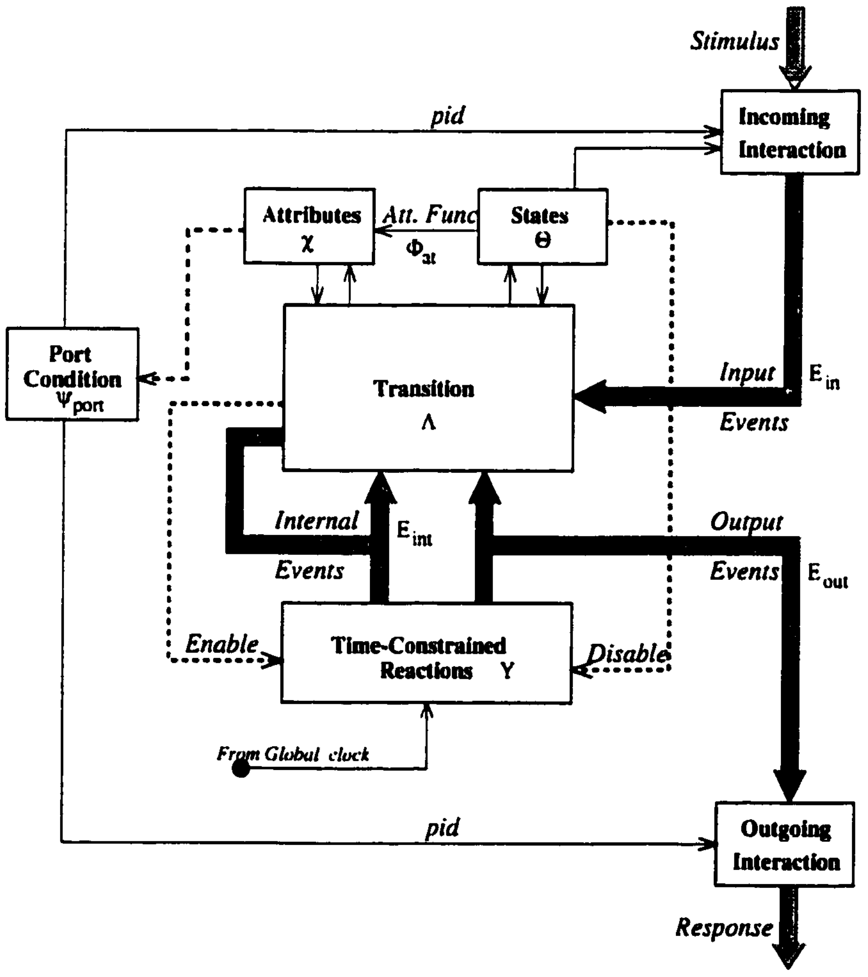


Figure 5: Anatomy of a TROM

has been discussed in [4] in the context of developing interface specifications for commercial class libraries. The functions in the interface tier specification for symbol table operations are shown in Figure 4. It has four operations which are explained below. For each operation, objects specified in the interface specification range over values denoted by the terms of sort *SymTab*.

- *init* - initializes the symbol table to be empty
- *insert* - modifies the table by adding a new binding to symbol table, if the key k is not in the domain of symbol table
- *lookup* - requires the key k in the mapping domain and returns the value of k and does not change the state of the symbol table i.e. new table is equivalent to the old table, and
- *delete* - requires the key k in the mapping domain and modifies the table by deleting the binding associated with k from the symbol table

For each operation, the **requires** and **ensures** clause specify its pre- and postconditions. A precondition on an operation is a predicate that must hold in the state on each invocation of the operation; if it does not hold, the operation's behavior in unspecified. A postcondition is a predicate that holds in the state upon return. An operation's clients are reponsible for satisfying preconditions, and its implementer is responsible for guaranteeing the postcondition. The **modifies** clause lists those objects whose value may possibly change as a result of executing the operation. Hence, *lookup* is not allowed to change the state of its symbol table argument whereas *insert* and *delete* are allowed to change. In the *insert* operation, we can remove the **requires** clause and use a special **signals** clause in its postcondition to specify that a signal should be raised if the key k is already found in the symbol table.

A TROM is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes, and timing constraints. A state in TROM can represent either a real-valued information of an environment object, such as the position of a moving train, or a system state. A state can be either simple or complex. In the later case, the state has substates, where each state can be either simple or complex. A TROM can include two kinds of attributes: port types, and LSL traits. The inclusion of an LSL trait brings in to the model the abstract data model and

```
TROM TROM_NAME [PORT_TYPE_PARAMS]
  Events: ⟨events⟩
  State: ⟨states⟩
  Attributes: ⟨porttypes⟩
  Attribute-function:
      states ↦ attributes
  Transition Spec:
      R_1 : ⟨source_state, destination_state⟩; event(port_condition);
              enabling_condition  ⟹ post_condition;
      ... :
      ... :
      R_N :
  Time-constraints:
      (TransitionSpec, event, [t_1, t_2], states)
      (...)
end
```

Figure 6: TROM Class Description

```
SCS SystemName
  Includes:
  Instantiate:
  Configure:
end
```

Figure 7: Template for System Configuration Specification (*SCS*)

consequently the abstract operations defined on it. A port has unique port-type and is an access point for bidirectional communication channel between a TROM and its environment(e.g. TROMs). A port-type determines the allowable messages at a particular port. A TROM can have multiple ports of same port-type and/or different port-types. An event is an activity that happens in an atomic interval of time while an action is an activity that happens in non-atomic interval of time (finite). An informal description of elements defined in a TROM object shown in Figure 5 are as follows:

- A set of **events** (E) partitioned into three sets namely, input events (E_{in}), output events (E_{out}), and internal events (E_{int}). Message passing is represented by the input and output events suffixed by the symbols ? and ! respectively.
- A set of **states** (Θ) in which a state can have many substates and the initial state is marked by the symbol *.

- A set of **attributes** (χ) in which the attributes could be of either an abstract data type specifying the *data model* or a port reference type.
- An **attribute-function** (Φ) that defines the association of attributes to states. If a computation is associated with a transition entering a state, only the attributes associated with that state can be modified and all the other attributes remain unchanged.
- A set of **transition specifications** (Λ) in which each specification describes the computational step associated with the occurrence of an event. A transition specification has three assertions, (i) *precondition*, (ii) *postcondition*, and a *port-condition* specifying a property of the port at which the event can occur. These assertions may involve attributes, and the keyword **pid** (port-identifier).
- A set of **time-constraints** (Υ) in which each time-constraint specifies the *reaction* associated with a transition. A reaction is the firing of an output or an internal event within a defined time period. A set of *disabling states* are associated with every reaction. When an object enters any of the disabling states of the reaction, an *enabled* reaction is disabled.

Figure 5 depicts abstractly the behavior of a generic TROM [1]. The flow of events are indicated by filled arrows. An input event is a result of an incoming interaction caused by external stimulus, the current state of TROM and the port with respect to port-condition. A computation is triggered by every event that updates the states and attributes determined by an attribute-function.

This computation enables a reaction which are indicated by the dashed arrows corresponding to time-constrained reaction. Similarly, an outstanding reaction may be disabled because of a state update. An outstanding reaction may be fired in the form of a transition based on the input from the global clock, thereby generating an internal or an output event. As a response to the stimulus, all the output events will result in the port specified by the port-condition.

In this model, port-conditions are used to specify the patterns of interactions between the components of a system. Typical real-time features such as minimal and maximal delays, exact occurrences, and periodicity of event occurrences, and combinations of temporal relations (stimulus-response, response-response, etc.) can also be specified. The model also provides encapsulation of timing constraints by

TROM *Train* [@C]
  Events: $Near!C, Exit!C, In, Out$
  State: $*S1, S2, S3, S4$
  Attributes: cr :@C
  Attribute-function:
    $S1, S3, S4 \mapsto \{\}$; $S2 \mapsto cr$;
  Transition Spec:
    $R_1 : \langle S1, S2\rangle$; $Near!(true); true \implies cr' =$ pid;
    $R_2 : \langle S2, S3\rangle$; In; $true \implies true$;
    $R_3 : \langle S3, S4\rangle$; Out; $true \implies true$;
    $R_4 : \langle S4, S1\rangle$; $Exit!$(pid $= cr$); $true \implies true$;
  Time-constraints:
    $(R_1, In, [3,5], \{\})$
    $(R_1, Exit, [0,8], \{\})$
end

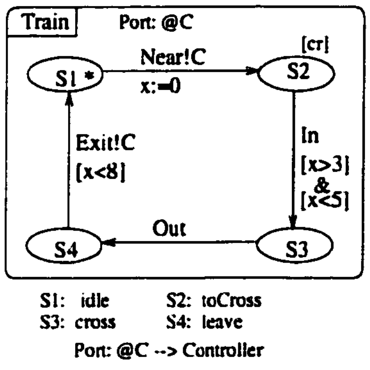


Figure 8: Class Specifications for Train.

precluding an input event from being a constrained event. In other words, a TROM cannot enforce any timing constraints on the occurrence of input events as they are under the control of the environment. See Figure 6 for the template sharing the language for a textual description of a TROM.

To summarize: A reactive object is described as in the textual description shown in Figure 6. We may regard this description as a specification of a reactive object. The name of the TROM followed by its elements should be presented for composing its specification. The modeling elements of a TROM are finite state machines, port types, LSL traits, and timing constraints. Each modeling element has a formal meaning. Consequently, a TROM description has an abstract internal description, called *Abstract Syntax Tree (AST)* [41]. From this internal representation, objects of a TROM class are obtained by instantiation. Since the characteristics of TROM classes are encapsulated in their definition, it is possible for a TROM class to inherit the characteristics of another TROM. The specification enviornment in TROMLAB permits the reuse of LSL traits from the library as well as composing them on the fly. In either case, it is the browser's responsibility to make them available. After composing a TROM from a set of modeling elements, it is submitted to the *Interpreter*. If it is syntactically correct, the browser may be invoked to save it in the reuse library. The relationship between a new TROM class, and LSL traits is automatically computed and updated in the reuse repository.

```
TROM Gate [@S]
  Events: Lower?S, Raise?S, Down, Up
  State: *G1, G2, G3, G4
  Transition Spec:
      R1 : ⟨G1, G2⟩; Lower?(true); true ⟹ true;
      R2 : ⟨G2, G3⟩; Down; true ⟹ true;
      R3 : ⟨G3, G4⟩; Raise?(true); true ⟹ true;
      R4 : ⟨G4, G1⟩; Up; true ⟹ true;
  Time-constraints:
      (R1, Down, [0, 2], {})
      (R3, Up, [2, 4], {})
end
```

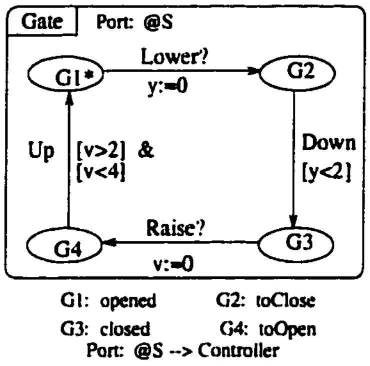


Figure 9: Class specifications for Gate.

4.3 SCS - System Configuration Specification

A *System Configuration Specification* (*SCS*) defines a subsystem composed of TROM objects. Figure 7 shows the template of *SCS*, which has *Include*, *Instantiate*, and *Configure* sections. Here <*name*> represents the name of a system being developed, and *Include* clause is used to import other subsystems. The *Instantiate* clause is used to define a reactive object by parametric substitutions to cardinality of ports for each port-type and initializing attributes (if any) in the initial state of the TROM. The objects specified in the *Instantiate* clause and the imported subsystems in the *Include* clause are composed to obtain a configuration through the *Configuration* clause. The communication links between compatible ports of interacting objects are set up through the composition operator $\leftrightarrow$. If the set of input message sequences at one port is a subset of the output message sequences at the other port, then the two ports are compatible and this relationship is symmetric.

A subsystem is composed only from objects instantiated from compiled TROM classes. It is also possible to include an already composed subsystem in defining a new subsystem. A subsystem is implemented by recursively aggregating all included subsystems, and suitably defining port links. In a subsystem, port links are established between TROM objects whereas in the included subsystems, the port links are established with the TROM objects. This facilitates the development of large subsystems as the available components can be reused as such. After composition, a subsystem may be saved by invoking the browser. All inherent relationships within a subsystem, and the relationships between a new subsystem and those subsystems in

```
TROM Controller [@P,@G]
  Events: Near?P, Exit?P, Lower!G, Raise!G
  State: *C1, C2, C3, C4
  Attributes: inSet : PSet;
  Traits: Set[@P, PSet]  /* Link to LSL tier */
  Attribute-function:
     C1 ⟼ {}; C2, C3, C4 ⟼ {inSet};
  Transition Spec:
     R1 : ⟨C1, C2⟩; Near?(true);
          true ⟹ inSet' = insert(pid, inSet);
     R2 : ⟨C2, C2⟩, ⟨C3, C3⟩; Near?(¬(pid ∈ inSet);
          true ⟹ inSet' = insert(pid, inSet);
     R3 : ⟨C2, C3⟩; Lower!(true); true ⟹ true;
     R4 : ⟨C3, C3⟩; Exit?(pid ∈ inSet);
          (size(inSet) > 1) ⟹ inSet' = delete(pid, inSet);
     R5 : ⟨C3, C4⟩; Exit?(pid ∈ inset);
          (size(inSet) = 1) ⟹ inSet' = delete(pid, inSet);
     R6 : ⟨C4, C1⟩; Raise!(true); true ⟹ true;
  Time-constraints:
     (R1, Lower, [0, 2], {})
     (R5, Raise, [0, 2], {})
end
```

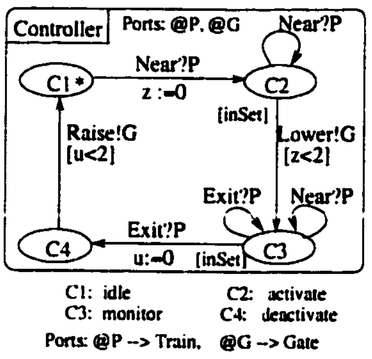


Figure 10: Class specifications for Controller.

the reuse library are automatically recomputed and updated in the reuse repository.

4.4 Train Gate Controller System - An Example

A generalized version of *Train-Gate-Controller* (TGC) problem has been used in TROMLAB simulation and verification [3]. To illustrate the reuse assistance that can be provided by the browser for applications based on trains and gates in transportation domain, we use the components from this example in building the test bed database for the browser.

In the TGC system, more than one train can cross a gate simultaneously from multiple parallel tracks. A train can independently choose the gate it will cross, based on its destination. So the interacting entities in the system are trains, controllers and gates. Figures 8, 9, 10 show the TROM class specifications of train, gate, and controller and their state machine descriptions.

In Figure 8, a train first sends the message *Near* (*Exit*) to a controller indicating that it is approaching (exiting) the gate. After sending the *Near* message, the train

```
SCS TrainGateSystem
  Include:
  Instantiate:
      $A_1, \ldots, A_m :: Train[@C : n].Create()$
      $B_1, \ldots, B_n :: Controller[@P : m, @G : 1].Create()$
      $C_1, \ldots, C_n :: Gate[@S : 1].Create()$
  Configure:
      $\forall i \in 1 \ldots m,\ j \in 1 \ldots n$
      $A_i.@c_j \rightarrow B_j.@p_i$
      $B_j.@g_1 \rightarrow C_j.@s_1$
end
```

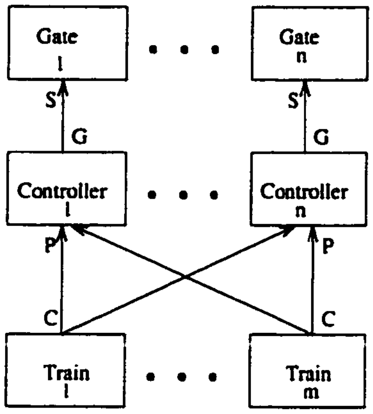


Figure 11: System configuration specification for Train-Gate-Controller system.

triggers an internal event *In* within a range of 3 to 5 time units and sends an *Exit* message within 8 time units. After receiving the *Near* message while in the *idle* state, a controller sends the message *Lower* (*Raise*) within 2 time units to the gate it is controlling, indicating that the gate has to be lowered (raised). The *Raise* message is then sent after receiving the *Exit* message from the last train leaving the crossing. After receiving the *Lower* message from the controller, the gate triggers an internal event *Down* within 2 time units and after receiving the *Raise* message triggers an internal event *Up* within a range of 2 to 4 time units. An important safety requirement is that the gate must be closed when a train is crossing the gate and that the gate must be monitered by a controller. Also, the gate must be eventually raised and must remain so for a certain period of time before it is lowered again to promote better operation of the system. The *SCS* of the TGC system is shown in Figure 11 and is obtained by composing the instances of TROM classes. For example, a system configured with three trains, two controllers, and two gates will have the following ports and links. That is, each train will have two ports of the same type referring to the controller; each controller will have two ports of the same type referring to the train and one port referring to gate; each gate will have one port referring to controller; one gate is linked to only one controller by linking the unique port of the gate to that port of controller referring to gate; each train is linked to both controllers by linking the two ports of that train to one port referring to train in each controller. Hence *SCS* permits configuration of a system in a succint manner. See Figure 12 for the port links in this system configuration. A set of m trains, n controllers and n

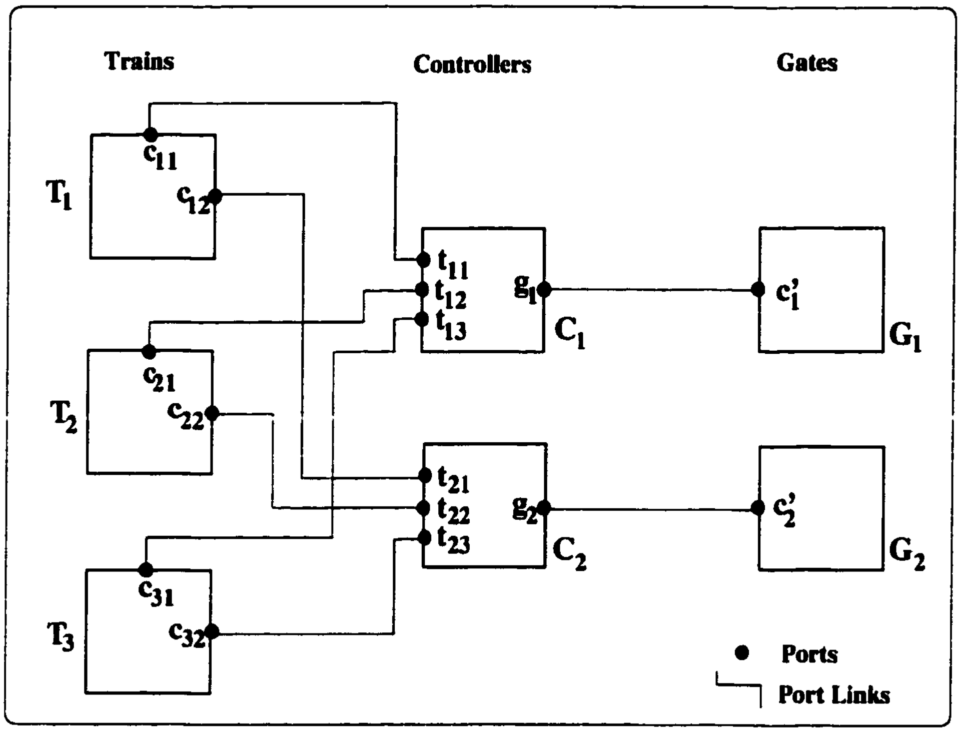


Figure 12: Train Gate Controller System with 3 Trains, 2 Controllers and 2 Gates

gates are composed as shown below:

1. Train Ports - T_i : $c_{i1}, c_{i2}, c_{i3}, \ldots, c_{in}$, $1 \leq i \leq m$

2. Controller Ports - C_i : $t_{i1}, t_{i2}, t_{i3}, \ldots, t_{im}$, $1 \leq i \leq n$

   : g_i, $1 \leq i \leq n$

3. Gate Ports - G_i : c'_i, $1 \leq i \leq n$

Therefore, the port links are between trains and controllers, and controllers and gates as below.

1. Link(T_i, C_j) : $c_{ij} \leftrightarrow t_{ji}$, $1 \leq i \leq m$; $1 \leq j \leq n$

2. Link(C_i, G_i) : $g_i \leftrightarrow c'_i$, $1 \leq i \leq n$

4.5 The Dynamics of Components

During the development of reactive systems, TROM specifications should be composed and compiled first and then can be followed by a composition of *SCS*. A TROM

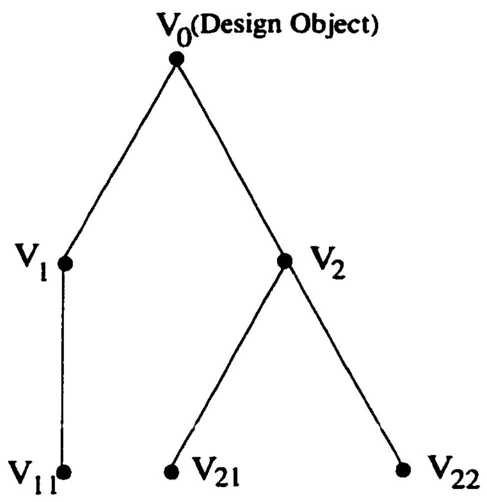


Figure 13: Version Hierarchy

specification may include one or more LSL traits. A TROM specification may be written from scratch or may reuse another TROM specifications. Similarly, LSL traits may either be reused from the LSL library or written from scratch. Consequently, all TROMLAB components undergo changes and versioning during the different phases of developing a reactive system. This section discusses in detail the dynamics of change of TROMLAB components.

A *version* is a semantically meaningful snapshot of a design component at a point in time. In general, the *versions* created for different applications have a hierarchical relationship as shown in Figure 13. A child component is a version of its parent; conversely, a child version is a derived version of its parent. A derived version may *inherit* from its parents. That is, V_1 may inherit V_0 and V_{22} may inherit V_2 but not directly V_0. Moreover, siblings do not have any relationship between them.

The LSL traits shown in Figures 14 and 16 are different versions of the LSL trait shown in Figure 3. These versions include new axioms, new functions and new interface operations. The main significance of the symbol table in Figure 3 is that it does not allow more than one occurrence of a (key, value) pair whereas this restriction is relaxed in the version shown in Figure 14. These two versions may be required in two different contexts and hence both are semantically meaningful. The trait in Figure 16 is another refinement of *SymTab*, which allows deletion of values of different occurrences for the same key. The function *find-all* in Figure 14 creates a list of all values having the same key. In order to allow the construction of tables having multiple values, the pre-condition for operation *insert* has been removed in the new version (Figure 15). Consequently, pairs of the form (1, 2), (1, 3), ... etc., can be added to the symbol table. The axioms for *isin* is still valid for the new version. However, its interpretation is that it returns true if the search key k_1 matches the most recent insertion of a pair (k, v), $k = k_1$. Consequently, no change in the interface function *lookup* is necessary.

The axiom for *rem* is not changed in $SymTab_2$ (Figure 16). Consequently, the delete operation in the interface specification (Figure 17) which uses *rem* in the post-condition ensures that the most recently inserted key in the symbol table is removed. Similarly, in Figure 17, the function *delete-all* removes a list of all values having the same key. The axiom for *rem-all* is added in the trait $SymTab_2$ (Figure 16) and the postconditions for delete operation is unchanged in Figure 17.

With these new interpretations, the traits $SymTab_1$ and $SymTab_2$ and their corresponding interface specifications may be regarded as refinements of *SymTab*. Their theories correspond to maintaining symbol tables arising in compilers for handling nested scoping rules in imperative programming languages such as Pascal.

We define a Larch/C++ interface specification C' to be a version of another Larch/C++ specification C if the **uses** clause of C' mentions an LSL trait which is a version of an LSL trait in the **uses** clause of C. This property can be checked from the syntax of interface specifications. Another type of version relation can be defined: the interface specification C' associated with the trait specification T' is a version of the interface specification C, if the trait T' is a version of the trait T' associated with C'.

Versioning between TROM classes can be defined in a multitude of ways. An example of versioning is shown in Figure 19. The *NewTrain* and *NewController* objects are instances of classes shown in Figure 18. These are versions of *OldTrain*, and *OldController*, which are instances of classes defined in Figures 8 and 10. For TROM class objects, a *version* is a descendent of some existing version (if not the first version) and can serve as an ancestor of additional versions in the form of hierarchical trees. New versions can also be created as part of the design process only if the changes that were made are consistent with the rules for inheritance [1] and are syntactically correct. It is the responsibility of the users to verify the inheritance properties.

If the new versions of train (*NewTrain*) and controller (*NewController1*) TROMs are used along with the old Gate TROM in defining a subsystem, a new *Train-Gate-Controller* system TCG1 will be created. We call TCG1 a *version* of the old subsystem TCG. Notice that replacing one component C of a system S_1 by its version C' leads to a version S_2 of the system S_1. Hence, a system may have several unrelated versions, and each version may give rise to several new versions. Consequently, managing subsystem versions poses greater challenge than managing TROM versions.

As discussed above, there are three kinds of design components. It is important to maintain and manage versions of each kind separately from other kinds. For composing new LSL specifications, only LSL traits and their versions will be required. A version of an LSL trait is allowed to add new signature and equations to the parent trait. All traits which are versions of a trait are grouped together to facilitate their viewing and reuse. However, versions and newly composed traits (totally unrelated to traits in the repository) are *not* part of the reuse repository. The versions of traits are managed hierarchically, and differently from the traits in the reuse repository. Separating those traits that have been verified to be semantically correct from those that are their versions (which may be syntactically correct) ensure both safety and flexibility for reuse. For example, the trait *Set* includes the two traits *Bool* and *Nat*. The traits *Set1* and *Set2* are refinements of *Set*. The trait *Set3* is a refinement of *Set2*. The traits *Set1*, *Set2*, and *Set3* may be included in three different TROM classes, see Figure 20. Similar design principles can be followed in composing subsystems. The tool manages versions of all components to assist their effective reuse.

$SymTab_1$: **trait**

includes *Bool*, *Set*

introduces
$emp :\rightarrow S$
$add : S, K, V \rightarrow S$
$rem : S, K \rightarrow S$
$find : S, K \rightarrow V$
$isin : S, K \rightarrow Bool$
$find\_all : S, K \rightarrow VL$

asserts
S **generated by** (emp, add)
S **partitioned by** $(find, isin)$
$\forall\ (s : S, k, k_1 : K, v : V)$
$rem(add(s, k, v), k_1) == \textbf{if } k = k_1 \textbf{ then } s \textbf{ else } add(rem(s, k_1), k, v)$
$find(add(s, k, v), k_1) == \textbf{if } k = k_1 \textbf{ then } v \textbf{ else } find(s, k_1)$
$isin(emp, k) == false$
$isin(add(s, k, v), k_1) == (k = k_1) \vee isin(s, k_1)$
$find\_all(add(s, k, v), k_1) == \textbf{if } k = k_1 \textbf{ then } \{v\} \cup find\_all(s, k_1) \textbf{ else}$
$find\_all(s, k_1)$

implies
converts $(rem, find, isin, find\_all)$ **exempting**
$(rem(emp), find(emp), find\_all(emp))$

$end SymTab_1$

Figure 14: LSL Specification of a New Symbol Table1

```
symbol_table1 is datatype based on S from SymTab1

init = proc() returns (s:symbol_table)
            ensures $s' = emp \wedge new(s)$

insert = proc(s:symbol_table, k:key, v:val)
    {
            modifies ($s$)
            ensures $s' = add(s, k, v)$
    }

lookup = proc(s:symbol_table, k:key) returns (v:val)
    {
            requires $isin(s, k)$
            ensures $v' = find(s, k)$
    }

lookup_all = proc(s:symbol_table, k:key) returns (vl:v_set)
    {
            requires $isin(s, k)$
            ensures $vl' = find\_all(s, k)$
    }

delete = proc(s:symbol_table, k:key)
    {
            requires $isin(s, k)$
            modifies ($s$)
            ensures $s' = rem(s, k)$
    }
end symbol_table1
```

Figure 15: Larch Interface Specification of a New Symbol Table1

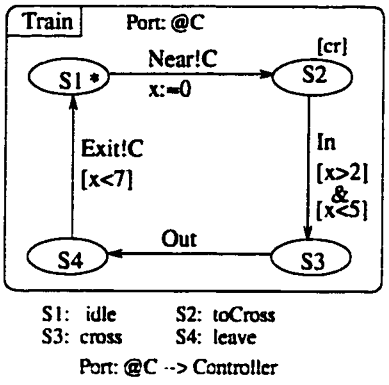
```
$SymTab_2$ : trait

    includes $Bool, Set$

    introduces
        $emp :\rightarrow S$
        $add : S, K, V \rightarrow S$
        $rem : S, K \rightarrow S$
        $find : S, K \rightarrow V$
        $isin : S, K \rightarrow Bool$
        $find\_all : S, K \rightarrow VL$
        $rem\_all : S, K \rightarrow S$

    asserts
        $S$ generated by $(emp, add)$
        $S$ partitioned by $(find, isin)$
        $\forall\ (s : S, k, k_1 : K, v : V)$
            $rem(add(s, k, v), k_1) == $ if $k = k_1$ then $s$ else $add(rem(s, k_1), k, v)$
            $find(add(s, k, v), k_1) == $ if $k = k_1$ then $v$ else $find(s, k_1)$
            $isin(emp, k) == false$
            $isin(add(s, k, v), k_1) == (k = k_1) \vee isin(s, k_1)$
            $find\_all(add(s, k, v), k_1) == $ if $k = k_1$ then $\{v\} \cup find\_all(s, k_1)$ else
                    $find\_all(s, k_1)$
                        $rem\_all(add(s, k, v), k_1) == $ if $k = k_1$ then
$s \cup add(rem\_all(s, k_1), k, \{v\})$
                    else $add(rem\_all(s, k_1), k, \{v\})$

    implies
        converts $(rem, find, isin, find\_all, rem\_all)$ exempting
                    $(rem(emp), find(emp), find\_all(emp), rem\_all(emp))$

$endSymTab_2$
```

Figure 16: LSL Specification of a New Symbol Table2

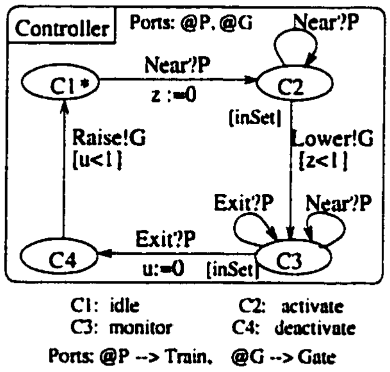
```
symbol_table2 is datatype based on S from SymTab2

init = proc( ) returns (s:symbol_table)
            ensures $s' = emp \wedge new(s)$

insert = proc(s:symbol_table, k:key, v:val)      {
            modifies $(s)$
            ensures $s' = add(s, k, v)$
    }

lookup = proc(s:symbol_table, k:key) returns (v:val)      {
            requires $isin(s, k)$
            ensures $v' = find(s, k)$
    }

lookup_all = proc(s:symbol_table, k:key) returns (vl:v_set)      {
            requires $isin(s, k)$
            ensures $vl' = find\_all(s, k)$
    }

delete = proc(s:symbol_table, k:key)      {
            requires $isin(s, k)$
            modifies $(s)$
            ensures $s' = rem(s, k)$
    }
delete_all = proc(s:symbol_table, k:key)      {
            requires $isin(s, k)$
            modifies $(s)$
            ensures $s' = rem(s, k)$
    }
end symbol_table2
```

Figure 17: Larch Interface Specification of a New Symbol Table2

TROM *Train* [@*C*]
Events: $Near!C, Exit!C, In, Out$
State: $*S1, S2, S3, S4$
Attributes: cr :@C
Attribute-function:
$S1, S3, S4 \mapsto \{\}$; $S2 \mapsto cr$;
Transition Spec:
$R_1 : \langle S1, S2 \rangle$; $Near!(true); true \implies cr' = \text{pid}$;
$R_2 : \langle S2, S3 \rangle$; In; $true \implies true$;
$R_3 : \langle S3, S4 \rangle$; Out; $true \implies true$;
$R_4 : \langle S4, S1 \rangle$; $Exit!(\text{pid} = cr)$; $true \implies true$;
Time-constraints:
$(R_1, In, [2, 4], \{\}), (R_1, Exit, [0, 6], \{\})$
end



TROM *Controller* [@*P*,@*G*]
Events: $Near?P, Exit?P, Lower!G, Raise!G$
State: $*C1, C2, C3, C4$
Attributes: $inSet : PSet$;
Traits: $Set[@P, PSet]$ /* Link to LSL tier */
Attribute-function:
$C1 \mapsto \{\}$; $C2, C3, C4 \mapsto \{inSet\}$;
Transition Spec:
$R_1 : \langle C1, C2 \rangle$; $Near?(true)$;
$true \implies inSet' = insert(\text{pid}, inSet)$;
$R_2 : \langle C2, C2 \rangle$, $\langle C3, C3 \rangle$; $Near?(\neg(\text{pid} \in inSet)$;
$true \implies inSet' = insert(\text{pid}, inSet)$;
$R_3 : \langle C2, C3 \rangle$; $Lower!(true)$; $true \implies true$;
$R_4 : \langle C3, C3 \rangle$; $Exit?(\text{pid} \in inSet)$;
$(size(inSet) > 1) \implies inSet' = delete(\text{pid}, inSet)$;
$R_5 : \langle C3, C4 \rangle$; $Exit?(\text{pid} \in inset)$;
$(size(inSet) = 1) \implies inSet' = delete(\text{pid}, inSet)$;
$R_6 : \langle C4, C1 \rangle$; $Raise!(true)$; $true \implies true$;
Time-constraints:
$(R_1, Lower, [0, 1], \{\}), (R_5, Raise, [0, 1], \{\})$
end



TROM *Gate* [@*S*]
Events: $Lower?S, Raise?S, Down, Up$
State: $*G1, G2, G3, G4$
Transition Spec:
$R_1 : \langle G1, G2 \rangle$; $Lower?(true)$; $true \implies true$;
$R_2 : \langle G2, G3 \rangle$; $Down$; $true \implies true$;
$R_3 : \langle G3, G4 \rangle$; $Raise?(true)$; $true \implies true$;
$R_4 : \langle G4, G1 \rangle$; Up; $true \implies true$;
Time-constraints:
$(R_1, Down, [0, 1], \{\}), (R_3, Up, [1, 2], \{\})$
end

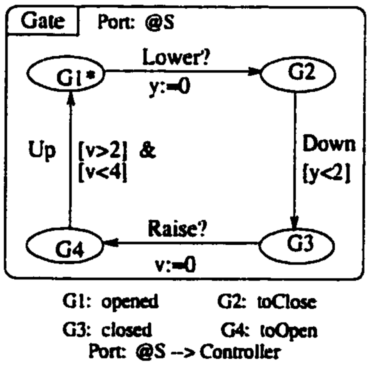


Figure 18: Class specifications for NewTrain, NewController and OldGate

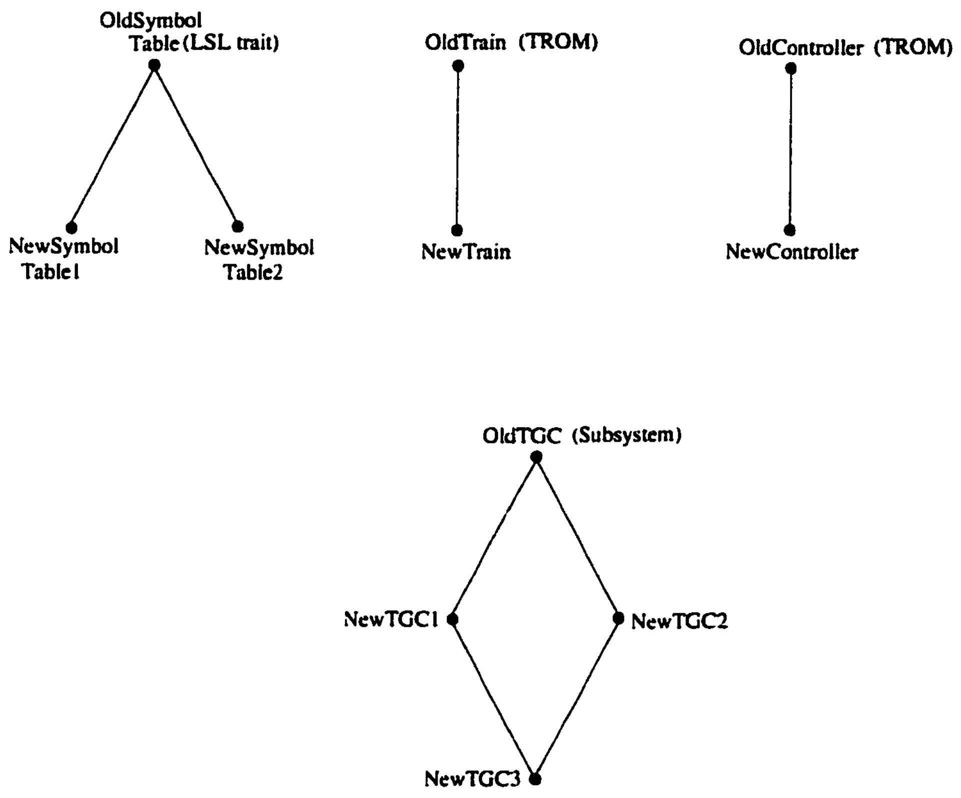


Figure 19: Versions in TROMLAB

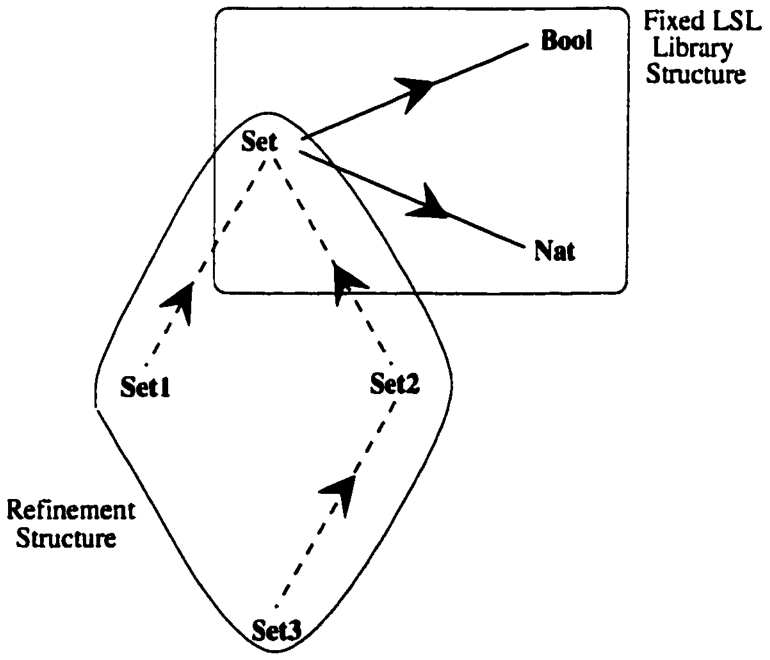


Figure 20: Version Hierarchy for Traits

Chapter 5

Designing the Browser

The three major functionalities of the browser are: *navigation and view*, *management of component repositories*, and *reuse promotion*. It can be invoked in two ways: through GUI of TROMLAB system, and directly through VISTA, the browser user interface. Navigation and viewing are accompanied by the associated activities such as querying, retrieving component information, redefining components, and recording redefined components in the reuse repository. Repository design for large-scale reuse of software components is a challenging task. We review some approaches in this chapter. We have decided to simplify the task, for reasons explained later, by adopting UNIX files and directories to comprise the database. The directory structures, and algorithms for retrieving reusable components and managing versions are described in the next chapter. The reuse phase composes new components using components from the library, invoking other software libraries and editors to assure completeness and consistency of the component. The browser design takes into account other criteria, such as providing good user interaction facilities at VISTA, in order to fulfill the above goals.

5.1 Component Repositories

The structure of a good repository is key to obtaining good retrieval results. If components are not indexed and structured properly, even intelligent retrieval algorithms will fail to produce satisafactory results. Currently, component-based software reuse faces a big dilemma: in order for the approach to be useful, the repository must be built effectively and populated with trusted components, and efficient retrieval techniques must be made available. Several retrieval techniques inherent to classical information retrieval technologies can be considered as viable candidates for finding relevant components. Methods based on formal specifications have been studied recently [8] for retrieving software components. These methods match signatures to retrieve a specification. But very little research has been done into the issues involving proper and effective design of databases for component-based reuse. Narayanan [31], has put forth an object-oriented database schema for the storage and retrieval of design patterns; however experience on using that approach has not yet been reported.

The retrieval methods used for software repositories can be divided into three categories: enumerated classification, faceted, and free-text indexing [12]. Hypertext systems also need some form of retrieval for navigation.

Enumerated classification is a well-known retrieval method used by *ACM Computing Reviews* Classification System. In this method, information is placed in categories that are structured in a hierarchy of subcategories, much like Unix file system. This scheme has the ability to iteratively divide information into smaller pieces that reduces the amount of information that needs to be viewed. The major drawback in this scheme is its inherent inflexibility and possibly difficulties in understanding large hierarchies. One should compromise between the depth of a classification hierarchy and the number of categories.

Most importantly, once the hierarchy is in place it gives just one view of the repository, namely the hierarchical division of categories. In order to use this repository, users must have a good understanding of the structure and contents of the repository. In particular, users must be able to distinguish between class labels; otherwise retrieval will be ineffective.

Faceted classification defines attribute classes that can be instantiated with different terms [33]. This is a slight variation of the relational model in which terms are grouped into a fixed number of mutually exclusive facets. Components may be searched by specifying one term for each facet. The number of facets used per domain is fixed, and within each facet classification methods can be provided to help users choose terms for retrieval. An advantage of this scheme over enumerated classification is that individual facets can be redesigned without affecting other facets. However, users must have an understanding of the significance of the facets, and the term that are used in the facet. It may become hard for users to find the right combination of terms that accurately describe the information needed, especially in large and complex information domains.

Free-text indexing (automatic indexing) methods use the text from a document for indexing. After removing fluffy words such as "and", and "the" from the extracted text, the rest of the text is used as an index to the document. Users specify a query using keywords that are applied to the indices to find matching documents. Although no classification effort is required, often human indexers augment the automatically extracted text to refine the index terms. Statistical measures are used to rank retrieved information [37]. These indexing schemes are most suitable for retrieving text documents, because they use linguistic terms and are easy to build and use. The low cost of building the repository coupled with their acceptable performance have made this scheme popular in commercial text retrieval systems and World Wide Web engines such as Yahoo or Alta Vista. Software components such as specification, design, and source code are nonlinguistic. Indexing methods can be made to work on the documented text of a software component. In this case, the retrieval is based on informal descriptions - it is not even dependent on the syntax of the component. Since effective component retrieval requires both the syntax and semantics of the components, indexing schemes are not applicable to retrieval of software components.

These methods define a spectrum from enumerated classification, which has a rich structure, to indexing, which requires no structure. In this spectrum, the cost and the difficulty of construction decrease from classification to indexing. It seems that good structure is necessary for effective retrieval. The important question not clearly answered in literature is the following: what is the relationship between structure and effectiveness of retrieval and can adequate retrieval effectiveness be accomplished with minimally structured repositories?

Very litle research has gone into answering the above question for the reuse of reactive system components. This thesis is the first attempt to put forth a reuse methodology within the framework of a browser. We are attempting to accomplish it in the context of TROMLAB components. Because of the experimental nature of this study, we follow a middle ground wherein the structure of the integrated environment, and information structures that are cheaply derived from the knowledge of the environment are effectively utilized. So, we use UNIX system directories of files produced at different stages of the software development process as our repository, and build indexes on top of them for retrieval of components. The resulting repository will be complete and consistent. Completeness is ensured by the processes and activities that produce and consume the components. We ensure consistency by separating the libraries of different types of components, and within each library we retain relationship among components and their versions. A key virtue of completeness and consistency is that information relevant to a query is always retrieved with "no confusion nor junk".

There is an inherent hierarchical structure, the one available for UNIX directories, for the repository. At each level of the hierarchy information in a file may be textual, or organized as a linked list or organized hierarchically. Consequently our retrieval algorithms are based on simple and effective algorithms that search and manipulate hierarchical and linked structures. An overall advantage of this approach is that the initial set-up cost is low and costs are incurred only incrementally on an as-needed basis. There is no extensive up-front cost, which would be unavoidable had we opted for a database approach. A database based approach, as used in faceted classification, is more general, more structured, and is more appropriate for a large-size reuse. However, desigining a schema for the repository and building inference engines for logical-based reasoning and retrieval of reusable components is a challenging research direction.

We have assumed that most of the users are aware of the formalisms, and languages governing the components. That is, users are also familiar with the classification hierarchy and contents of the repository. VISTA is designed to help users in understanding the formalism of the system, structure and behavior of design components, and TROMLAB features.

During the development phase, new components (not related to components in the repository) may be constructed. Several versions of a component existing in the repository may be under development. The developer can invoke the version control mechanism in the tool to manage these components in the repository. For example, when the developer creates a new LSL trait, it is recorded as an original version. If it is modified later in deriving a new trait, the refined trait is recorded as a version in the reuse library. Versioning of TROM and subsystem components are also controlled by the tool in a similar manner. We discuss version control of components in a later section.

As remarked earlier, components of the same type may be related, and there may exist relationship between components of different types. For example, a trait may be **included** in many traits. This is a static relationship between LSL traits. A trait may be used in several Larch/C++ specifications. This is a static relationship between components of two different types. A trait may be an attribute of several TROMs. This is another relationship between components of different types. Some of these relationships may be used to define new relationships: two TROMs are related if they share a trait. A dynamic relationship arises when subsystems are constructed from TROM instances. These relationships are recorded in the browser tool and are used in navigation algorithms.

5.2 Navigation and View

The search engines on the repository structures are buried under VISTA, which facilitates navigation, view, and other tasks. Navigation is a guided tour, a metaphor borrowed from *hypertext*. Rather than confined to a linear order of documents, users are able to move through a hypertext document by following links represented on the window by buttons. The basic building blocks of a hypertext are *nodes* and *links*. Each node is associated with a unit of information. Nodes can be of different types. For example, node type may depend on the data type stored in a node (text, graphics, audio, specification, program) or the domain type whose objects are stored in a node (programs, LCPPs, LSLs, TROMs). Links define relationships between nodes.

For example, a link can connect a trait in LSL with a TROM indicating that the trait is an attribute of the TROM object. Some links are directed, and some are bidirectional (symmetrical relationship), and some links can be transitively traversed. A guided tour constrained by the direction of navigation in a link enhances the power of navigation. When confronted with a large number of navigational possibilities, the guided tour approach usually will constrain to a few important choices. Components can be inspected by moving from one component type library to another and within each repository taking a guided tour offered by VISTA. These navigational capabilities facilitate rapid traversal of the repositories to locate target objects for reuse. Since links are set up based on the semantic information of components, navigation is helpful in zeroing in on the component with a required functionality. Whenever the semantics of a TROMLAB component changes, it should be recorded and the links changed to reflect the change. As the information on the components undergo changes, and the number of components in the repository varies, both nodes (components in repositories), and links (semantic relationship) must be modified. A software developer can become disoriented if the resulting hypertext network is too complex and the navigation produces too many choices. This complexity is avoided by localising the guided tours: the guided tours are non-intersecting and within each guided tour only components of one type can be viewed and navigated.

Navigation built on hypertext metaphor seems most suitable for retrieving reusable components. Developers can actively refine their search by inspecting software components and deciding the direction for further navigation. That is, developers are in control of the identification process, and consequently will be able to conduct a thorough search. The interactive nature of navigation and component inspection will result in a better understanding of the repository contents. This in turn reduces search time and cost of search in future endeavour. The hypertext based search can withstand the growth of repository contents. Components within each type can be added without affecting the search performance. New component types can also be added without affecting the quality of search. The benefits of this navigational aid provided in VISTA will have a favorable impact on the potential for reusing components in TROMLAB environment.

5.3 Reuse Promotion

Software reuse in the context of repositories of components can be accomplished in two ways: (i) when an application makes a call to a component that already exists in the repository, and (ii) when a developer designs an application by making a call to a component that was developed in the context of another application, and, as a result, has been stored in the repository. The browser enables reuse of the second instance; that is, the focus is on reuse that occurs during software construction, where software developers experience the greatest difficulties in finding reusable software products.

Reuse of components is usually preceded by an exploration of the set of candidates from the repository. This is accomplished by querying and navigation. VISTA provides these capabilities. When the identification process is concluded, the user will have located and retrieved a small set of applicable components that can be reused. More importantly, this process will yield information about whether there are components with the appropriate functionality that are in the repository. This decision will assist the developer in deciding whether or not to build a component from scratch. Navigating through the basic components may also lead to viewing versions of the components, which have already been tried in different applications. The various guided tours group together components which have *similar* but distinct functionality. Developers can easily inspect similar objects by following these guided tours.

Some of the retrieved components may not be relevant to the construction task and the developer may wish to refine some of these components to fit the task. Using the editing facilities offered by VISTA, a retrieved component may be edited. After a new component is obtained, VISTA facilities for syntax checking should be invoked before recording changes to the components in their repositories. The developer is responsible for the semantics of the component; it can be tested by invoking LP for LSL components, and the simulator for TROM components.

5.4 Browser User Interface

VISTA provides textual and graphical descriptions of the LSL traits, Larch/C++ specifications, TROM classes and subsystems and will display their informal descriptions in a multiple window environment. Components can be queried, viewed, loaded, edited, and compiled in an easy interactive mode. When a user queries about a particular trait, VISTA displays the type along with the newly derived versions of that type, textual description of the respective traits and graphical version history tree (hierarchy). This will indeed enable the user to find the best trait needed for the application being developed or choose one that will require minimal change to achieve the required functionality. For example, a software designer looking for OO class specifications only has to know the types of classes available, and the types of messages each object of a class can receive and act upon. In such a case, the designer has only need to determine what objects are required, what the various object types should do and how the various objects or versions of objects relate to each other.

VISTA provides facilities for accomplishing these tasks.

VISTA provides high-level editing operations, and it provides templates for constructing LSL and Larch/C++ specifications. The menu and button interface enables the user to load a specification component into the editor allocate area, and compose into a more complex specification. There are buttons in the edit window that permit the user to create a new specification, submit a specification for compilation, store a specification, and invoke Larch software tools.

VISTA can be invoked in two modes: (i) as part of the TROMLAB environment, or (ii) as a stand alone browser. When invoked from TROMLAB environment, the following tasks may be performed:

- The user can create a new LSL trait or modify an existing trait from the reuse library to include in the TROM classes or subsystems.
- The user can choose to search for a particular LSL trait, TROM class, or a subsystem using keywords such as data types, trait names, class names, TROM name, subsystem name and specify relationships among components.
- The user can navigate through the entire set of stored information space.

When used as a stand alone browser, VISTA offers the following features, which are restricted to Larch library:

- A complete directory listing of LSL trait filenames and LCPP specification filenames available in the library database are provided side-by-side on the window.
- The filenames are hypertexts and has the capability of opening up the corresponding file upon a mouse click.
- The user may open (LSL trait or LCPP) a file by a mouse click:
  - The respective hypermode hierarchy tree of traits/classes at different levels of abstraction is displayed in a small window (hierarchical hypertext trees: hierarchy for traits and inheritance tree for classes). This will not interrupt the interface communication between the opened file and the user.
  - When the hypertext traitname in the tree is clicked, a window is opened with the display of corresponding trait file. If the chosen trait has **includes**

or **assumes** clauses defined, then the trait window also has hypertext links to them and can be viewed by a mouse click.

  - The hierarchy level of the hypertext tree may be restricted to 5 traits/classes. This reduces the flood of information which may lead to confusion.

- The user can choose to search for traits/classes using keywords which can be data type names, trait names, class names.

- The closest possible matches to the query will be displayed in the hypertext mode and then the user can repeat from any item to further probe the specifications.

- If no formal specification exist, then the user will be asked if a new specification should be constructed. In this case, the user is provided with the LSL (Larch/C++) specification template and a C++ code development area to verify its correctness. The user may request the newly created class to be added to the repository.

- The user can modify an existing class to suit the needs of a new application and add to the existing database as a new class.

- Components can be added to existing hierarchies and new hierachies can be constructed.

The above functionalities are consistent and complete with respect to the requirements of VISTA stated in Chapter 3.

5.5 Architecture

The design components of VISTA are shown in Figure 21. To support modularity and extendibility of the tool, we followed the OO design principles. The tool consists of the following classes; a *Workspace Manager (WM)*, an *Object Manager (OM)*, a *Query Handler (QH)*, a repository of LSL traits, a repository of LSL versions, a repository of *Larch/C++ Versions*, and a *Visual Interface for Software Reuse of* TROMLAB *Applications*(VISTA). *QH* is supported by *QY_LSL*, a query handler for LSL traits, *QY_LCPP*, a query handler for Larch/C++ specifications, and *QY_TR_SCS*, a query handler for TROMs and subsystems.

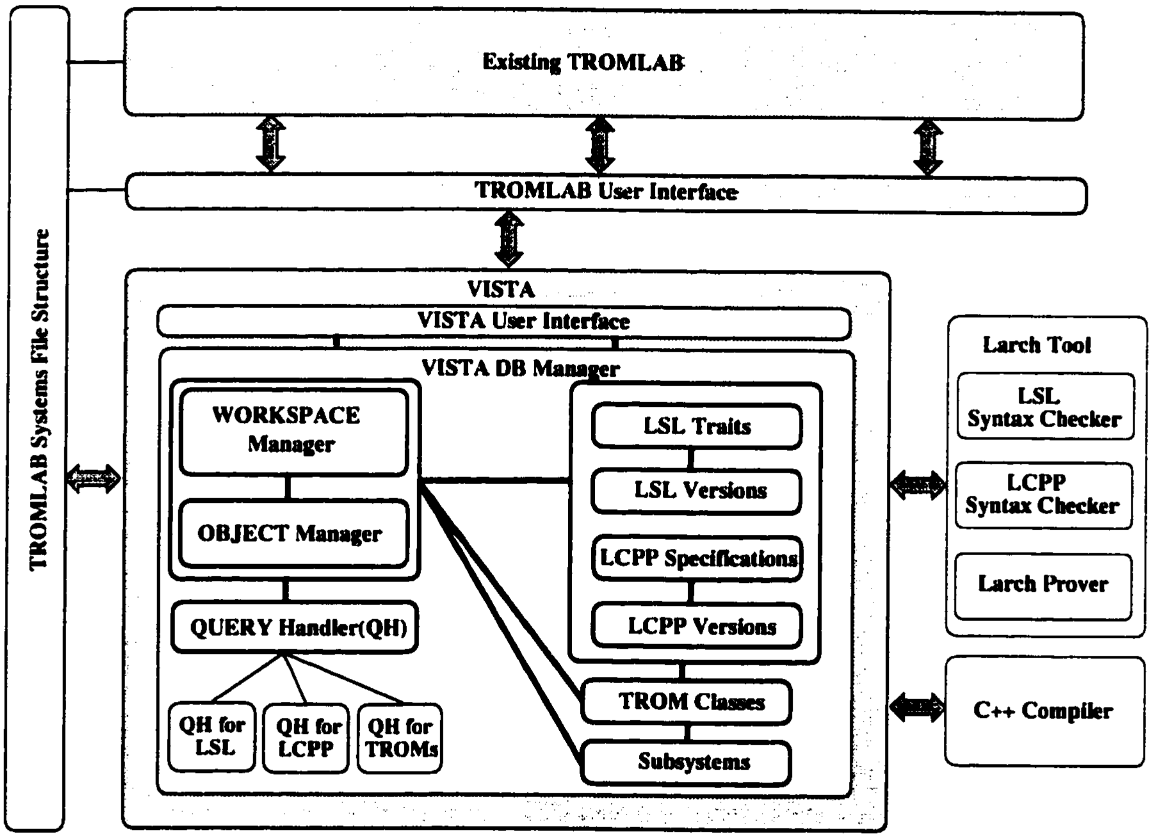


Figure 21: Architecture of VISTA

- **Workspace Manager -** This component manages the allocation and deallocation of specification workspace (current version of the LSL trait to be developed) of the browser. *WM* is invoked by the Object Manager for the purpose of editing files. Whenever a file is to be edited, the name of the file and a reference to the file are communicated to the Workspace Manager. Upon receiving this information, the Workspace Manager allocates a temporary buffer where in the contents of the files are copied. It also allocates an editor for editing the buffer. At the completion of the editing session, the Workspace Manager saves the file, either by overwriting or by creating a new file as commanded by the user, and returns the file to the Object Manager.

- **Object Manager -** It maintains and manipulates versions of all TROMLAB components. GUI invokes the Object Manager whenever a file has to be inspected or edited. The Object Manager initiates the search for the file. If the search is successful, the reference to the file is then passed on to the Workspace Manager. The Query Handler can also invoke the Object Manager by specifying a filename or certain keywords. The keywords may refer to the names of the traits, TROMs and or Subsystems.

- **Query Handler -** This process handles all the queries about TROMLAB components in the database and has three subclasses, QH for traits, QH for LCPP and QH for TROM classes. Query Handler can initiate the following type of queries from a user.

  1. List the names of subsystems.
  2. List the names of available TROM classes.
  3. List the LSL traits used by a particular TROM.
  4. List the LSL traits of a particular type (original as well as derived).
  5. List the hierarchy of *versions* of a particular type of TROM systems or LSL traits.
  6. List the hierarchy tree of included TROM subsystems within a system, and LSL traits included in each TROM along with its textual description. For each trait, the version hierarchy can also be requested.

7. Display the textual description of components; if graphical descriptions are requested, VISTA invokes GUI, and GUI takes over the dialogue with the user.

QH invokes the query handler that is specific to one component type and returns the results to *WM*, who in turn displays the results to the user.

- LSL **Traits Repository** - This is a directory of UNIX files of Larch traits. Some of them are taken from Larch library [15]; however, a majority of them were developed at Concordia University [4] in the context of developing specifications for Rogue Wave Library C++ [36] classes. All traits have been checked for syntactic and semantic correctness. Traits developed by users are maintained in a separate file; however, they are also part of the reuse library.

- LSL **Traits Versions** - Many traits in LSL repository have different versions which have similar but distinct behavior. The versions are maintained in a hierarchy, where the root of every hierarchy is a trait in the LSL repository. It is up to the user to declare whether a refined Larch trait is a version. No semantic checking is made to ascertain the logical relationship between versions. The user can make use of LP to check for desirable properties in a trait before declaring it as a version.

- **LCPP Repository** - This repository contains Larch/C++ interface specifications for Rogue Wave Library *tools.h++* [36]. They have been type checked by the Larch/C++ tool. These components can be included in the simulation tool during the validation phase.

- **LCPP Versions** - A Larch/C++ specification can be a subtype of another Larch/C++ specification [9, 22, 25]. The user should use Larch/C++ tools to ascertain such a relationship. The browser maintains versions of Larch/C++ specification in a hierarchical structure.

- **Visual Interface for Software Reuse of** TROMLAB **Applications** - VISTA provides graphical representations of the hierarchy trees, provides information on the components in the repositories, enables querying, and navigation of the TROMLAB components.

There is a dialog between the VISTA and GUI of the TROMLAB system at the following instances.

- a user wants to fetch an LSL trait or an LCPP from the Larch/C++ Reuse Library
- a user wants to save a *version* of the trait, or Larch/C++ specification.
- a user wants to inspect (save) a TROM class object or a subsystem.

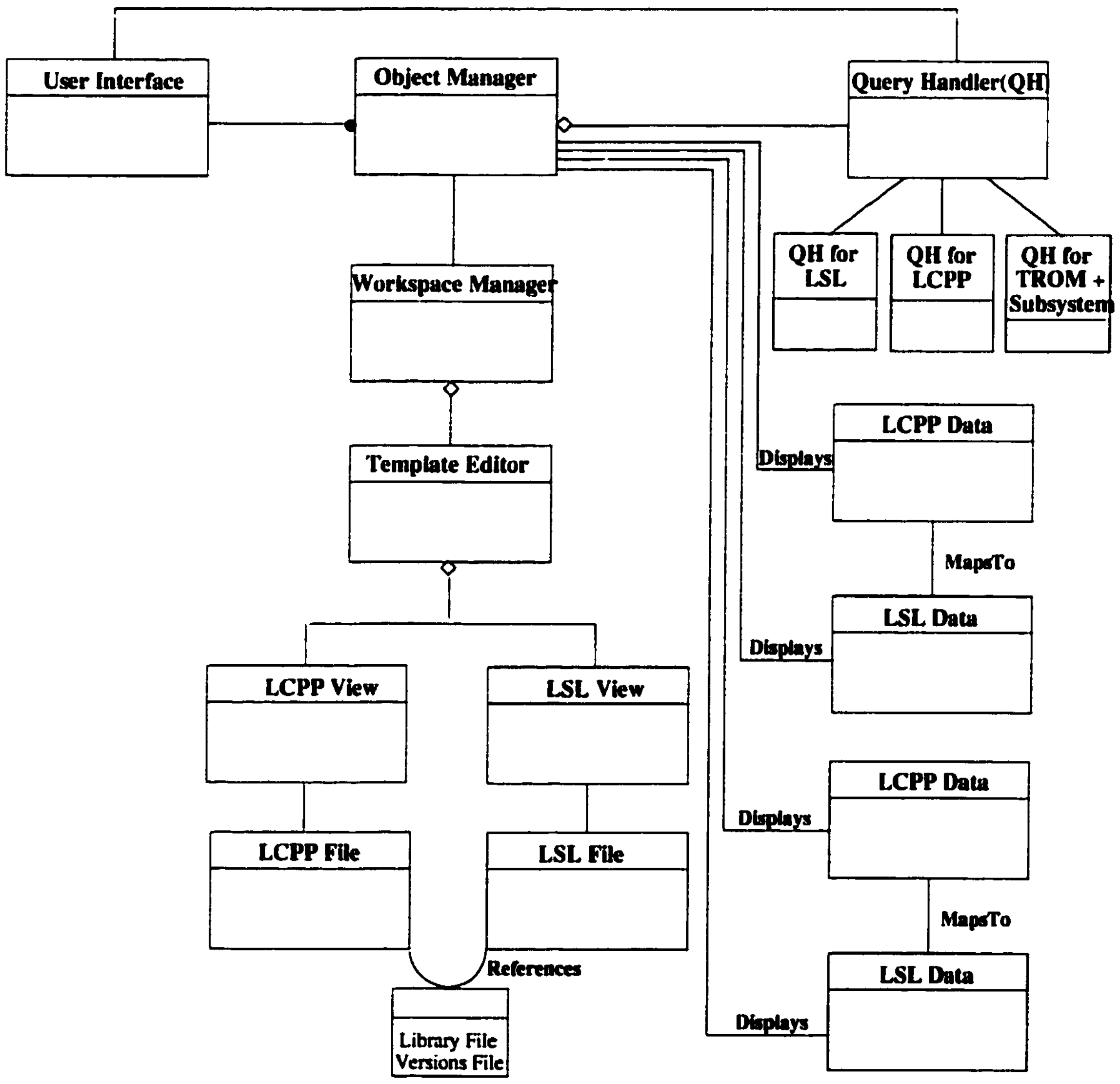


Figure 22: OMT Object Model

The (OMT) object model and class diagrams with interfaces for the design components of VISTA are shown in Figures 22, 23.

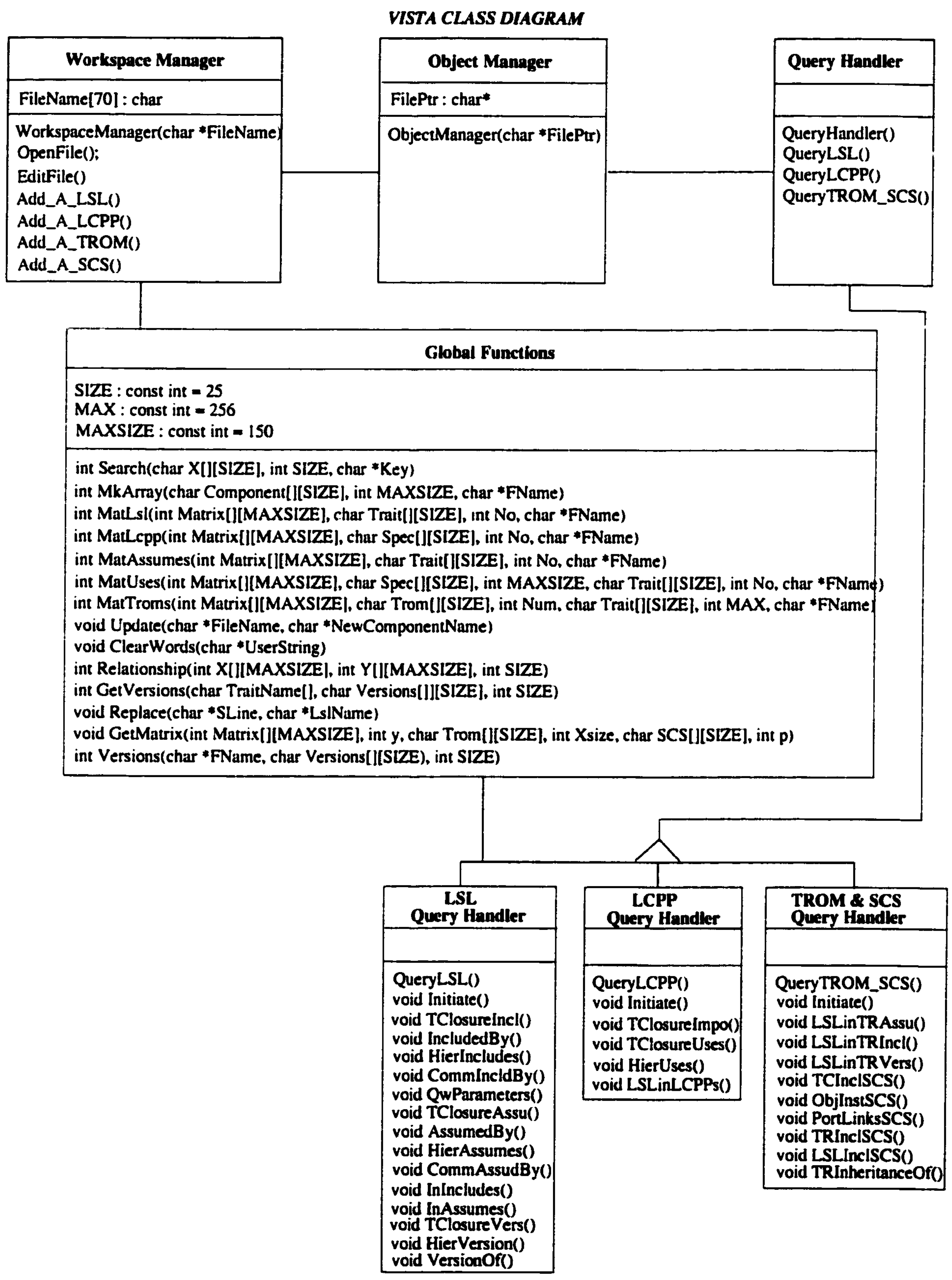

VISTA CLASS DIAGRAM
Workspace Manager
FileName[70] : char
WorkspaceManager(char *FileName)
OpenFile();
EditFile()
Add_A_LSL()
Add_A_LCPP()
Add_A_TROM()
Add_A_SCS()
Object Manager
FilePtr : char*
ObjectManager(char *FilePtr)
Query Handler
QueryHandler()
QueryLSL()
QueryLCPP()
QueryTROM_SCS()
Global Functions
SIZE : const int = 25
MAX : const int = 256
MAXSIZE : const int = 150
int Search(char X[][SIZE], int SIZE, char *Key)
int MkArray(char Component[][SIZE], int MAXSIZE, char *FName)
int MatLsl(int Matrix[][MAXSIZE], char Trait[][SIZE], int No, char *FName)
int MatLcpp(int Matrix[][MAXSIZE], char Spec[][SIZE], int No, char *FName)
int MatAssumes(int Matrix[][MAXSIZE], char Trait[][SIZE], int No, char *FName)
int MatUses(int Matrix[][MAXSIZE], char Spec[][SIZE], int MAXSIZE, char Trait[][SIZE], int No, char *FName)
int MatTroms(int Matrix[][MAXSIZE], char Trom[][SIZE], int Num, char Trait[][SIZE], int MAX, char *FName
void Update(char *FileName, char *NewComponentName)
void ClearWords(char *UserString)
int Relationship(int X[][MAXSIZE], int Y[][MAXSIZE], int SIZE)
int GetVersions(char TraitName[], char Versions[]][SIZE], int SIZE)
void Replace(char *SLine, char *LslName)
void GetMatrix(int Matrix[][MAXSIZE], int y, char Trom[][SIZE], int Xsize, char SCS[][SIZE], int p)
int Versions(char *FName, char Versions[][SIZE), int SIZE)
LSL
Query Handler
QueryLSL()
void Initiate()
void TClosureIncl()
void IncludedBy()
void HierIncludes()
void CommIncldBy()
void QwParameters()
void TClosureAssu()
void AssumedBy()
void HierAssumes()
void CommAssudBy()
void InIncludes()
void InAssumes()
void TClosureVers()
void HierVersion()
void VersionOf()
LCPP
Query Handler
QueryLCPP()
void Initiate()
void TClosureImpo()
void TClosureUses()
void HierUses()
void LSLinLCPPs()
TROM & SCS
Query Handler
QueryTROM_SCS()
void Initiate()
void LSLinTRAssu()
void LSLinTRIncl()
void LSLinTRVers()
void TCInclSCS()
void ObjInstSCS()
void PortLinksSCS()
void TRInclSCS()
void LSLInclSCS()
void TRInheritanceOf()

Figure 23: OMT Detailed Class Diagram

5.6 Overview of VISTA

When the browser is invoked as a stand alone tool, the main window opens and offers the user four options along with buttons to invoke external Larch tools and a C++ compiler:

- "Files"
- "Query"
- "C++ Compiler" - if user would like to develop code for the formal specifications he/she created using VISTA
- "Exit"
- "Help"

See Figure 35. By clicking on one of the options, the user can exercise the functionalities in that option. When clicked on the "Help" button, the system provides a glossary of technical terms, references to their usage, simple examples, and a brief summary to VISTA. If the "Exit" button is clicked, the system gives two options: to exit from TROMLAB system or enter into GUI; and to save unsaved files. If the "Query" button is clicked, the user is given five options; see Figure 52:

- Query LSL traits
- Query LCPP files
- Query TROMs and subsystems
- Exit
- Close (to go back to the previous window)

If the "Files" option is made in the main menu, the system opens up a window (Figure 36) in which there are four options to choose from:

- LSL files
- LCPP files

- TROM files
- Subsystem files

By clicking on any one of these options, the system responds with a menu having two options: Library files, and Version files. When an option is clicked, the system opens up a window which contains the names of files in the chosen collection. For example, if LSL files was chosen followed by "Versions", all LSL version files will be on the screen within the window. This window, shown in Figure has three buttons "Inspect", "Edit", "New" and "Cancel". When a file name is selected in the window and the button "Inspect" is clicked, the contents of the file is opened in a separate window (see Figure 40) for inspection. If the button "New" is clicked, space is allocated in a new window and an editor is assigned for composing a component. The "Cancel" button cancels the previously chosen option. If the "Edit" button is clicked, either after inspecting or before clicking on "Inspect" button, the browser allocates an editor to the file, and the file can be edited in a new window allocated for this purpose. However, no LSL file from the main library can be edited; it can only be viewed, copied, reused in a design. In general, this window has "Save", "Save As", "Update DB", "Close", and "Exit" buttons. These buttons have the following functionalities:

- "Save" - overwrites the file.
- "Save As" - the file will be saved under a new name given by the user.
- "Close" - closes the session, but confirms any opened unsaved file.
- "Exit" - closes the window and exits from the window to the previous menu.
- "Update DB" - the update options differ for different components; for example, a trait may become a version of another trait, a TROM object may become part of a subsystem, or a trait may be included in a TROM. The functionality of this button is to maintain the database behind VISTA to manage the TROMLAB components.

The user can perform "Update DB" on LSL versions. The "Update" operation on TROMs and subsystems are done by the Object Manager at the request of GUI. This is because, semantic checking on TROM components are done by TROMLAB components that interact with GUI.

We explain in the next chapter the window structures, query input facilities and viewing facilities of query reponses and a complete prototype description is provided in Chapter 7.

5.6.1 Interaction of VISTA with TROMLAB Modules

VISTA interacts through GUI, the graphical user interface tool of TROMLAB, with some of the components of TROMLAB. A TROM system comprises of TROM classes which include LSL traits and TROM subsystems. The editor in GUI interacts with VISTA to fetch the TROM classes and subsystems stored in the VISTA's repositories. When a requested component is not available in the repository, WM allocates storage and an editor to compose the TROM component. This can be done from scratch or by invoking the browser again to retrieve suitable components for reuse. After composition and syntax analysis, a syntactically correct component may be saved by invoking the browser. The user will be required to give full information on the type of component and its status (version). The browser determines the relationship of the new component to other components in the repository and records the relationship in the repository.

Both *Interpreter*, and the *Simulation Tool* interact with the browser to record changes to TROM class specifications during the process of simulation, and debugging in the *debugger mode*. To specify the behavior of simulated actions the user interface may interact with the browser to fetch Larch/C++ specifications from the reuse library. Whenever subsystems are required for a simulation run, the browser may be invoked to retrieve a subsystem based on information such as included TROM names, and port links.

If the system is initialized for simulation in the *default mode*, the *interpreter* fetches the necessary TROM classes, subsystem configurations, and LSL traits from the browser and proceeds with the simulation. Once the system is initialized for simulation in the *debugger mode*, the *Interpreter* recompiles TROM classes, and subsystem configurations, and the browser ensures that LSL traits are syntactically correct, and then resumes simulation. There is also a provision in the browser to invoke C++ compiler. This feature will be used in future when the TROMLAB environment is extended with a code generation environment. The Larch/C++ classes provide the framework to develop semantically correct code.

Chapter 6

Reusing TROMLAB Components

A prototype reuse search system to operate within TROMLAB object repositories has been built. In this chapter, we discuss the contents and structure of reuse repositories, give the basic algorithms for navigation and query processing, and present the features of VISTA visual interface features.

6.1 Storage Structures

The repository is a collection five directories: **LSL**, **LCPP**, **Subsystems** and **TROMS**, **VISTA-C++**. The directory **LSL** contains all relevant "*.tex" and "*.lsl" trait files; **LCPP** contains all "*.lcc.tex" and "*.lcc" files for all the LCPP Specifications; **VISTA-C++** has five directories **Stage_1**, **Stage_2**, **Stage_3**, **Stage_4** and **Stage_5**; **TROMS** contains all the TROM files. **Subsystems** contains all the Subsystem files.

The directory **Stage_1** has in turn two directories: **Assumes** and **Includes**. The directory **Assumes** contains five programming files for the queries of **assumes** relation (see below) and three data files. The **Includes** directory contains six programming files and three data files.

1. **Assumes** directory
   (a) Data file descriptions:
      - **assumes.dat** contains the set of tuples, where each tuple is a trait in the assumes relation followed by the traits related to it by the assumes relation.

- `trait_ass.dat` contains only those LSL traits which have non-empty assumes relation.
- `closure.dat` contains the (0,1) matrix representation of the assumes relation.

(b) Programming file descriptions:

- `hierachy.cpp` which on execution gives the hierachy representation of the assumes relation for a given LSL trait.
- `intersection.cpp` which computes the common LSL traits that two LSLtraits assumes.
- `trait_exist.cpp` which checks if the given trait is already in the assumes relation or not.
- `matrix.cpp` which on execution gives the transitive closure of the assumes relation and also the total number of traits in the assumes relation for a given LSL trait.
- `assumed.cpp` which on execution gives the number of traits and the list of trait names assumed by a given trait.

2. **Includes** directory

(a) Data file descriptions:

- `InputFile.dat` contains tuples, where each tuple is a trait in the includes relation followed by the traits related to it by the includes relation.
- `TraitFile.dat` contains all LSL traits for which the includes relation is non-empty.
- `matrix.dat` contains the (0,1) matrix representation of the includes relation.

(b) Programming file descriptions:

- `h_tree.cpp` which on execution gives the hierachy representation of the includes relation for a given LSL trait.
- `common.cpp` which gives the common LSL traits included by two LSL traits.

- `relation.cpp` which on execution gives the transitive closure of the includes relation for a given LSL trait and also the total number of traits it includes.
- `paramtr.cpp` which on execution with a trait in the form A(p for X, q for Y) returns the filename of the trait matching the trait substituted with the given actual parameters.
- `existence.cpp` which on execution checks if the given trait is already in the trait file of the includes relation or not.
- `included.cpp` which on execution gives the number of traits and the list of traits a given trait includes.

The directory `Stage_2` has two sub-directories: `Imports` and `Uses`. The `Imports` directory contains one programming file and three data files. The `Uses` directory contains three programming files and two data files.

1. `Imports` directory
   (a) Data file descriptions:
      - `Imports.dat` contains tuples, where each tuple is a Larch/C++ specification followed by Larch/C++ specifications related to it by the imports relation.
      - `LSpecifications.dat` contains all the LCPP Specifications having non-empty imports relation.
      - `Impo_matrix.dat` contains the (0,1) matrix representation of the imports relation.

   (b) Programming file description:
      - `Reln_imports.cpp` which on execution gives the transitive closure of the imports relation for a given LCPP Specification.
2. `Uses` directory
   (a) Data file descriptions:
      - `uses.dat` contains the LCPP Specifications in the uses relation and the related LSL traits.

- `LSpecifications.dat` contains LCPP Specifications which have non-empty uses relation.

(b) Programming file descriptions:

- `Reln_uses.cpp` which on execution gives the transitive closure of the uses relation for a given LCPP Specification.
- `Hierachy_uses.cpp` which on execution gives the hierachy representation of the uses relation for a given LCPP Specification.
- `Used_by.cpp` which on execution gives the list and the number of all LCCP Specifications which uses a given LSL trait.

The directory `Stage_3` has the version files for LSL traits and also contains the programming files for the queries of "version of" relation and one data file. The data file `Version.dat` contains tuples, where each tuple consists of a trait followed by its versions in the "version of" relationship. There are three programming files:

- `query_version_of.cpp` which on execution with a trait A and a version V of A will find the versions of V in the hierarchy rooted at A.

- `version_closure.cpp` which on execution with a trait A, and a version V of A will find the transitive closure of the version V in the "version of" relation . When the version V is not given, the program outputs all the versions of A.

- `hierachy_versions.cpp` which on execution with a trait A and one of its version V will find the hierachy of the version V. If the version is not given, then the hierachy of versions for the trait A is given.

The directory `Stage_4` has the programming files for updating version files of LSL traits. It uses data files from *Stage_3* directory. The programming files are:

- `Addition_v.cpp` which on execution with a trait A and a new version V1 of V, adds version V1 as a version of V in the hierarchy rooted at A. If the version V is not given then V1 is added as a new version of A.

- `Deletion_v.cpp` which on execution with a trait A, a version V1 of version V, deletes version V1. If version V is not given, then V1 is assumed to be the first level version of the trait A and is deleted from "Version.dat" file.

- Add_a_trait.cpp which on execution adds a newly composed LSL trait to the "Version.dat" file, by checking if it is a valid LSL trait and it also creates a version file for that LSL.

At present the repository does not have version files for Larch/C++ specifications. When they become available, they will be organized and manipulated similar to the way LSL traits have been organized.

The directory Stage_5 has two directories TROMS and SCS. It contains eleven programming files and five data files. The data files are:

- SCS.dat contains subsystem names.
- SCS_TROMS.dat contains tuples, where each tuple is a subsystem followed by a list of instantiated TROM objects.
- SCS_DATA.dat contains tuples, where each tuple is a subsystem followed by subsystems included in configuring it.
- TROMS.dat contains the list of TROM names.
- Trom_data.dat contains tuples, where each tuple is a TROM name, followed by a list of parameters and a list of traits it includes.

The programming files are:

- h_assumes.cpp which on execution gives the hierachy relationship of the assumes relation of the LSL traits included in a given TROM.
- h_includes.cpp which on execution gives the hierachy relationship of the includes relation of the LSL traits that a given TROM includes.
- h_versions.cpp which on execution gives the hierachy relationship of the"version of" relation of the LSL traits that a given TROM includes.
- Query_2.cpp which on execution gives the transitive closure of the includes relationship for a given subsystem.
- Query_3.cpp which on execution determines all the TROM objects instantiated in a given subsystem.

- Query_4.cpp which on execution shows the port links in a subsystem.
- Query_5.cpp which on execution gives all the subsystems which include a given TROM.
- Query_6.cpp which on execution determines all the traits included in the objects within the given subsystem.
- Add_a_SCS.cpp which on execution adds a given TROM file to the directory of TROMS.
- Add_a_TROM.cpp which on execution adds a given subsystem file to the directory SCS.
- Query_version.cpp which on execution with a TROM, which is an inheritance of another TROM, gives the TROM name in the "inheritance of" relationship and also the type of the inheritance (either behavioural, extensional or polymorphic).

6.2 Larch Reuse Environment

When VISTA is invoked to compose a new LSL trait, an LSL *Development Window* is provided to the user. Both LSL and LCPP development windows have basically the same functionality; the differences will be noted. See Figures 40, 46.

The *Development Window* consists of a srollable editable specification text area, and a non-editable file path text field. The *Specification* text field indicates the corresponding LCPP specification file of the currently loaded LSL trait into the *Development Window*. The full directory of the path name of the selected file is shown. The scrollable text area is where the text of the specification is developed. As explained in the previous chapter, only traits in the version repository can be modified. When the modification or composition of the specification is complete, the user can submit it for syntax checking, and then save the file before quitting. The options on the window are used to achieve the following functionalities:

- Loading a known specification.
- Composing a new specification.
- Editing a specification loaded from version repository.

- Checking the syntax of a trait. The output of the syntax checker is displayed in a text window.
- Checking the axioms in a trait using LP. The Larch Prover interface results are displayed in a text window.
- Saving a file after editing; the file is saved under the same file name.
- Saving a file with a new file name.
- Exiting to the main window.

Browsing and inspecting a selected trait can be done independent of any development activity. An important feature of VISTA is the provision of *templates* for LSL traits. The template shown in the *Development Window* is the LSL template. Each section of the template is labeled by a keyword. The user must enter the keyword and the punctuations in the respective sections before entering the text appropriate to the section. The template feature allows the user to edit any of the scrolled areas of the specification. The "Reset" button may be clicked at any instant to return the specification to the state it was in when originally loaded into the template. LSL traits have two relationships: *includes* and *assumes*. The *includes* section lists all traits that are necessary for composing the current trait. The semantics of an included trait is that the signature, equations and theories implied by it and those of all the traits transitively included in it are imported into the current specification. The semantics of *assumes* is slightly weaker: there is a proof obligation, in the sense that any trait that imports a trait with an assumes clause cannot have the implied theories automatically included; they need to be proved. The "Help" facility of the browser provides on-line implementation of the semantics.

Both *includes* and *assumes* relations are irreflexive, antisymmetric and transitive. The transitive closure of *includes*(*assumes*) can be computed using Warshall's algorithm [10]. This closure includes all pairs (A, B) such that A *includes* (*assumes*) B. Similar statement is true for *assumes*. During design, several *versions* of an LSL trait may be created. The relation "A is a *version* of B" is reflexive, antisymmetric and not transitive.

In Figure 24, B is a *version* of A, C is a *version* of B, D, E are *versions* of C and F is a *version* of D and E. Both *includes* and *assumes* are static relationships.

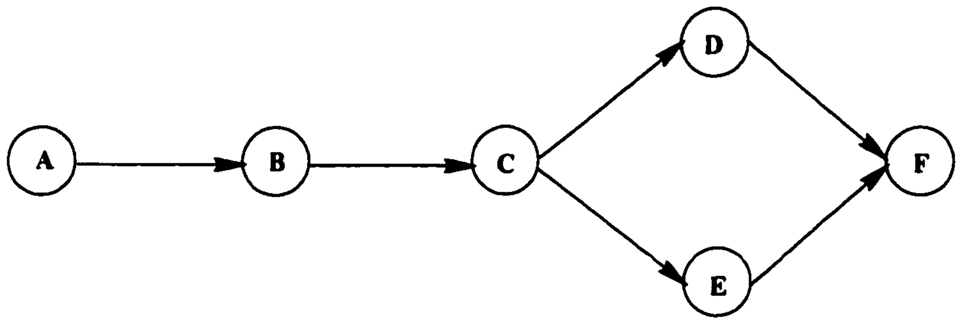


Figure 24: DAG of Versions

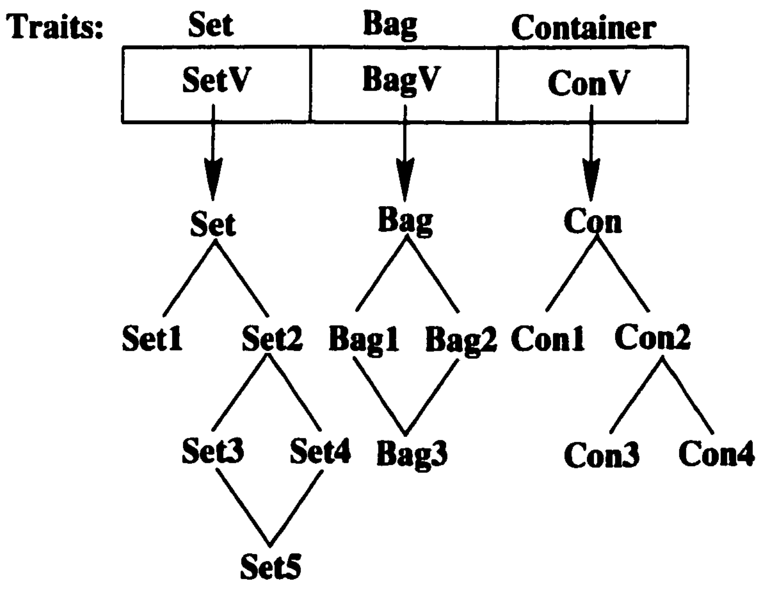


Figure 25: Version List of reusable LSL traits

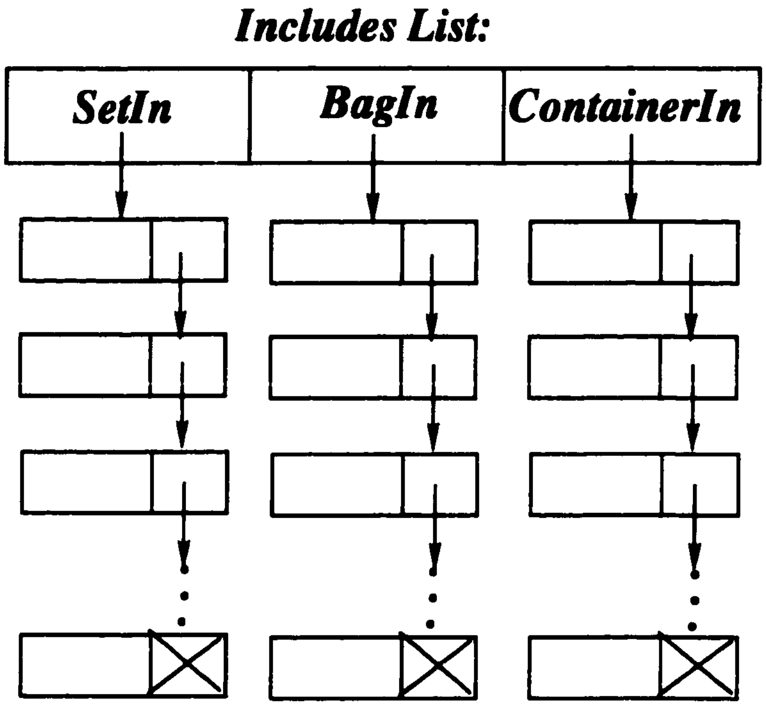


Figure 26: **includes** List of LSL traits

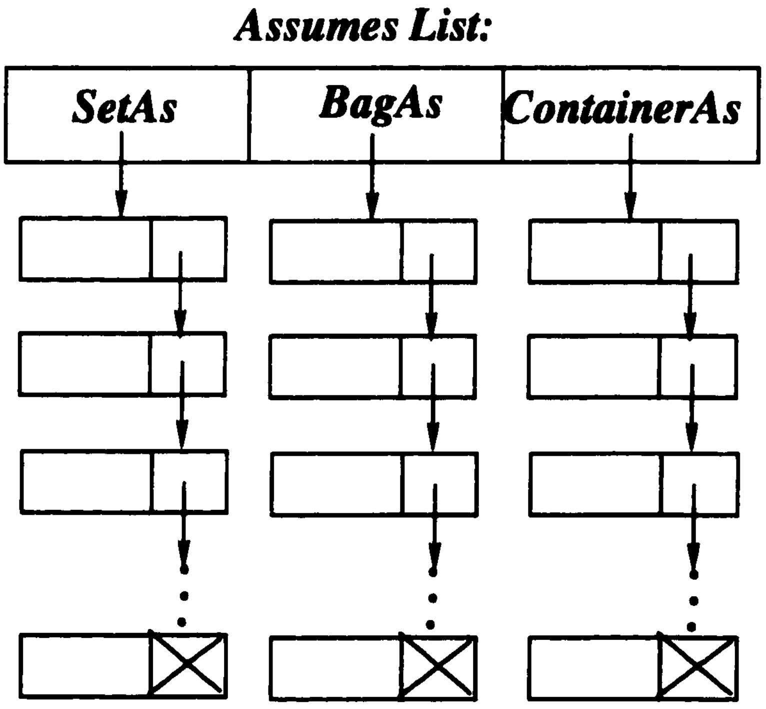


Figure 27: **assumes** List of LSL traits

File.lsl	Version List	Includes List	Assumes List
Set	**SetV**	**SetIn**	**SetAs**
Bag	**BagV**	**BagIn**	**ConAs**
Con	**ConV**	**ContIn**	**BagAs**
•	•	•	•
•	•	•	•

Figure 28: Table as a Data Structure

However, the *version* relationship is dynamic. For each trait in the library, several *versions* as shown in Figure 25 may exist. The *version* files are not part of the reuse library. However, each LSL *version* will have its own *includes* list, *assumes* list, and *version* list. If we were to maintain the *includes* list, *assumes* list, *version* list for each LSL trait in machine memory, the data structures shown in Figures 25, 26, 27, 28 would be appropriate. Figures 26 and 27 show the template for a collection of linked lists for three LSL traits Set, Bag and Container. The table in Figure 28 has four entries per row, where each entry is associated with an LSL trait: The first entry is the name of a trait; the second entry is a pointer to the *version* list of the trait; the third entry is a pointer to the *includes* list; and the fourth entry is a pointer to the *assumes* list. Figure 29 illustrates a comprehensive storage organization for a sample set of reusable components. The data structures in these four figures also portray the various files that we maintain in the repositories and their interrelationships.

Whenever a *version* Y of the LSL trait X is created, Y is inserted in the *versions* list of X, an entry for Y is made in the table and pointer to the *includes* and *assumes* list are recorded in the table. Note that the files themselves are stored in their respective directories and only pointers are kept in the table. Whenever a *version* Y of X is

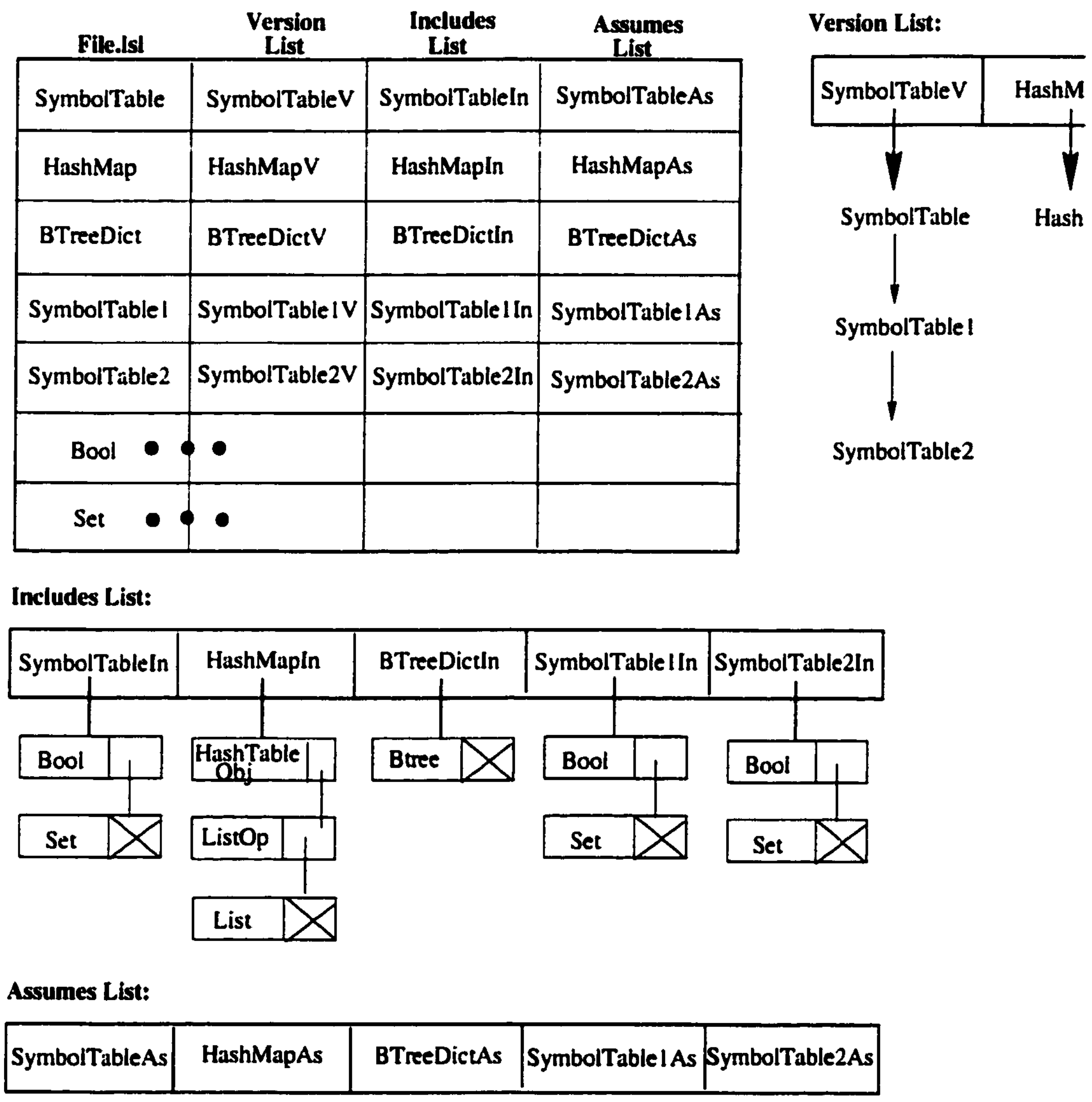


Figure 29: An Example Data Store of 3 Files

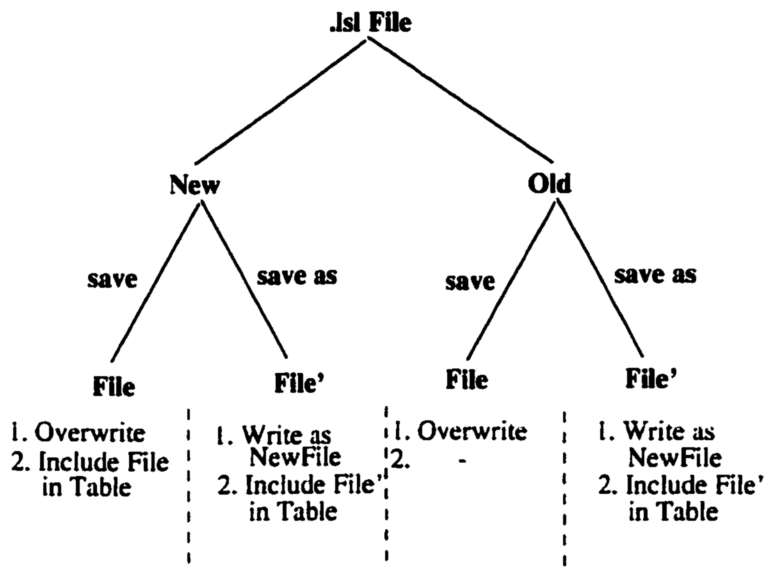


Figure 30: Notion of Save

deleted, then the following changes are made to the data structures.

- Y is deleted from the *version* list of X.
- The entry for Y from the table is removed.

These relationships among LSL traits may be viewed at any time by querying the system. The different contexts for saving an LSL file is shown in Figure 30.

A *Query Session Window* as shown in Figure 52 may be created at any time by the user to query the properties of the specification in the repository. In this window, there is a radio button corresponding to each query. A query is stated in natural language against the button. By clicking on a button, the user is asking for information against that query. The system opens up a dialog box for the user to input the parameters required to process the query. Upon completing the dialog box information sections, the user must click on "Apply" button to activate the search methods for retrieving the information from the repository. The system response is displayed in a text window. The following sequence of dialogs leads to query selection:

1. If the user chooses **Query on LSL traits**, the user is asked to choose one of the following:
   - Query on includes or assumes relation of LSL traits
   - Query on version relationship on LSL traits

- Exit
- Close - go to previous menu

2. If `Query on includes or assumes relation of LSL traits` is chosen, the user is asked to choose one of the following options:

    - Query on includes relation
    - Query on assumes relation
    - General information (Statistics)
    - Exit
    - Close - go to previous menu

3. `Query on includes relation` - The following queries are answered here:

    - **Query-1** Find the transitive closure for the *includes* relation. The number of traits included in a particular trait is also given as part of the response.

      The member function *Query_1()* of the class **QH_LSL** processes the above query.

    - **Query-2** Find all the LSL traits that *includes* a particular trait and also the number of traits. This query requests the *included by* relation. The member function *Query_2()* of the class **QH_LSL** processes the above query.

    - **Query-3** Find the hierachy relation for each trait in the *includes* relation. The member function *Query_3()* of the class **QH_LSL** processes the above query.

    - **Query-4** Find the common traits included by two given traits. The member function *Query_4()* of the class **QH_LSL** processes the above query.

    - **Query-5** Given a trait name with parameters, replace the existing parameters in the trait file and put the modified file in a temporary file.

      The member function *Query_5()* of the class **QH_LSL** processes the above query.

4. `Query on assumes relation` - The following queries are answered here:

- **Query-1** Find the transitive closure for the *assumes* relation. It also gives the number of traits a particular trait assumes .

  The member function *Query_6()* of the class **QH_LSL** processes the above query.

- **Query-2** Find all the LSL traits that *assumes* a particular trait and also the number of traits. It requests the *assumed by* relation.

  The member function *Query_7()* of the class **QH_LSL** processes the above query.

- **Query-3** Find the hierachy relation for the trait in the *assumes* relation.

  The member function *Query_8()* of the class **QH_LSL** processes the above query.

- **Query-4** Find the common LSL traits assumed by two given traits.

  The member function *Query_9()* of the class **QH_LSL** processes the above query.

5. `General Information` - The following queries are answered here:

- **Query-1** Find if the given trait is already in the trait file of the *includes* relation or not.

  The member function *Query_10()* of the class **QH_LSL** processes the above query.

- **Query-2** Find if the given trait is already in the trait file of the *assumes* relation or not.

  The member function *Query_11()* of the class **QH_LSL** processes the above query for*assumes* relation.

- **Query-3** Find the number of traits that *includes* or *assumes* a particular trait.

  The member function *Query_1()* of the class **QH_LSL** processes the above query for *includes* relation and the member function *Query_6()* of the class **QH_LSL** processes the above query for *assumes* relation.

- **Query-4** Find the number of traits *included by* or *assumed by* a particular trait.

The member function *Query_2()* of the class **QH_LSL** processes the above query for *included by* relation and the member function *Query_7()* of the class **QH_LSL** processes the above query for *assumed by* relation .

6. `Query on version relationship on` LSL `traits` - The following queries are answered here:

   - **Query-1** Find the transitive closure of the version V of trait A in the *version of* relationship.

     The member function *Query_12()* of the class **QH_LSL** processes the above query.

   - **Query-2** Find the hierachy of the version V , of a given trait A. If the version V is omitted find all the versions of A.

     The member function *Query_13()* of the class **QH_LSL** processes the above query.

   - **Query-3** Given A, the root of a version tree, and a version V of A, find all the versions of A.

     The member function *Query_14()* of the class **QH_LSL** processes the above query.

6.2.1 How queries are answered?

The data structure shown in Figure 28 is a good description of LSL files had they been stored internally in machine memory. For an understanding of query processing, this diagrammatic representation is helpful. See Appendix A for the full path names of directories and files. The table in Figure 28, which portrays the relationship of files in these directories, consists of a main index of LSL traits. Each row has three indexes where the first is an index list of *version* filenames of the LSL trait, the second one is a list of *includes* clause of filenames and the third one is a list of *assumes* clause of filenames. The main index has a flag which indicates whether the file is protected or unprotected. The protected files are those that are in the LSL *reuse library* and the unprotected files are the *versions* or new files created by users. So whenever the user wants to browse a file, the *includes* list of hyperlink file names are provided for the *includes* clause along with the *version* filename links, if any. Similarly, the *assumes* list of hyperlink file names are provided for the *assumes* clause along with the *version* filename links. The three relationships *assumes*, *includes*, and *version* have same abstract properties: *irreflexive, antisymmetric, transitive.* We use Warshall's algorithm [10] to compute the transitive closure of a generic relation represented by a (0,1) matrix, and instantiate this to realize the traits in the three respective relationships. These relationships are precomputed; they will be updated automatically after changes occur in the version repository. Based on the file structures given in Appendix A, and the precomputed relations, the queries are handled. An informal description of query handling methods is described below:

Querying the LSL library

- **Query-1** Given a trait A to output all the traits and the number of traits which A includes.

  The query is answered as follows:
  We search the file *matrix.dat* for trait A, and display the traits corresponding to the value 1 in the row of trait A. The programming file *matrix.cpp* in the directory "VISTA-C++/Stage_1/assumes" computes the response to the query for the *assumes** relation and the programming file *relation.cpp* in the directory "VISTA-C++/Stage_1/includes" answers the above query for the *includes** relation.

- **Query-2** Given a trait A to output all the traits and the number of traits included by A.

  The query is answered as follows:
  We search the file *matrix.dat* for trait A, and display the traits corresponding to the value 1 in the column of trait A. The programming file *assumed.cpp* in the directory "VISTA-C++/Stage_1/assumes"computes the response to the query for the *assumes** relation and the programming file *included.cpp* in the directory "VISTA-C++/Stage_1/includes" answers the above query for the *includes** relation.

- **Query-3** Given a trait A to check if the given trait is in the trait file of the (includes/assumes) relation.

  The query is answered as follows:
  We search the file *trait_ass.dat* for trait A, and display if the trait exists or not.

The programming file *trait_exist.cpp* in the directory "VISTA-C++/Stage_1/assumes" computes the response to the query for the *assumes** relation and the programming file *existence.cpp* in the directory "VISTA-C++/Stage_1/includes" answers the above query for the *includes** relation.

- **Query-4** Given two traits,to find the included traits common to them.

  The query is answered as follows:
  We search the file *matrix.dat* and find the transitive closure of both the traits and then find the traits common to them. The programming file *intersection.cpp* in the directory "VISTA-C++/Stage_1/assumes" answers the above query for the *assumes** relation. The programming file *common.cpp* in the directory "VISTA-C++/Stage_1/includes" responds to the above query for the *includes** relation.

- **Query-5** Given a trait in the form A(p for X,q for Y), it is required to return the filename matching the trait substituted with the given actual parameters.

  The query is answered as follows:
  We search the *InputFile.dat* to see if the trait exists with the given number of parameters. If a match is not found return an error message saying that the trait is incompatible with the given number of parameters, or does not exist. However, if a match is found a copy of the trait's file is made in a temporary file and the parameters p for X and q for Y, are substituted in the text of the trait and the user is provided with the temporary filename. The programming file *paramtr.cpp* in the directory "VISTA-C++/Stage_1/includes" answers the above query for the *includes* relation.

Querying the Versions

- **Query-1** Given a trait A and a version V of A to find the corresponding version in the *version of* relationship.

  The query is answered as follows:
  The query can be answered only if the version V is not in the first level because the solution for this query should only be a version and not the name of the trait. The version file for the given trait A is opened and the (0,1) matrix for the versions is formed and then the versions that V was derived from are

sent to the screen. The programming file *query_version_of.cpp* in the directory "VISTA-C++/Stage_3" processes the above query.

- **Query-2** Given a trait A, and a version V of A , find the transitive closure of the version V in the *version of* relation . If the version is not given ,then output all the versions of A.

  The query is answered as follows:
  If both A and V are given, the program outputs all the versions transitively included in V. However if only A is given, then for each version of A (in the first level as given in *Version.dat* file), we compute and output all versions transitively included in it. The programming file *version_closure.cpp* in the directory "VISTA-C++/Stage_3" processes the above query.

- **Query-3** Given a trait A and one of its version V find the hierachy of the version V. If the version is not given, then display the hierachy of versions for trait A.

  The query is answered as follows:
  In order to find the first level versions of A , the file *Version.dat* is used and the hierachy for those versions are obtained from the version file of the trait. If the version is given then only version file of the trait is used. The programming file *hierachy_versions.cpp* in the directory "VISTA-C++/Stage_3" processes the above query.

Updates to LSL Versions

It is possible to add or delete a version to the version file of the given trait. A file consisting of a set of traits and their first level versions are in the file *Version.dat* in the directory "VISTA-C++/Stage_4". Each LSL trait has it's own version file as explained earlier. All the version files are in the directory "VISTA-C++/Stage_4".

- **Query-1** Given a trait A, a new version V1 can be added as a version of V. If V is not given then V1 is added as a new version of A.

  The query is answered as follows:
  The version file for the given trait is opened and the (0,1) matrix for the version relation is formed . The new version V1 is added as a version of the already

existing version V and then the corresponding changes are made to the version file of the trait. If the version V is not given then it is added as a new version to the version file. The programming file *Addition_v.cpp* in the directory "VISTA-C++/Stage_4" processes the above query.

- **Query-2** Given a trait A delete the version V1 which is a *version of* V. If V is not given, then V1 is assumed to be the first level version of the trait A.

  The query is answered as follows:
  The version file for the given trait is opened and the (0,1) matrix for the version relation is formed . The version V1 which is the version of the already existing version, V is deleted and then the corresponding changes are made to the version file of the trait. If the version V is not given then changes are made to the *Version.dat* file. The programming file *Deletion_v.cpp* in the directory "VISTA-C++/Stage_4" processes the above query.

- **Query-3** To add a new trait A to the *Version.dat* file.

  The query is answered as follows:
  The given trait is validated to check if it is a trait. If it is not already present in the *Version.dat* file then it is added to the file. The version file for the trait is also formed. The programming file *Add_a_trait.cpp* in the directory"VISTA-C++/Stage_4" processes the above query.

6.3 Managing LCPP Files

The *Specification Development* window for LCPP specifications has features similar to the LSL development window. The major distinction is the template structure. The LCPP template is shown in Figure 46. When the template is displayed in "Edit" mode, any of the scrolled text areas can be edited. Functions are listed in the *Functions* list. To view or edit a function, double click on that function in the list, and the function template will appear. After editing the text in the function template, click "Apply" to record the change and close the function template. By clicking "Reset", instead of "Apply", the text is not changed. The button "Close" can be used at any time to close the function template without making change. To create a new member function of a class, click on "New". A function template will be opened. After entering the text, click "Apply". The new function is added to the function list in *Functions.*

An interface specification allows one or more other specifications to be imported. As an example, the interface specification RWZone shown in Figure 31 **imports typedefs**, another Larch/C++ interface specification. RWZone is a partial specification for the Rogue Wave description of Zone classes [36].

The language does not allow an interface specification to import itself. Moreover, if a specification A **imports** specification B and the specification B **imports** another specification C then A **imports** C. This transitive relationship is not permitted to include cycles. As explained in the previous section, this relationship and its transitive closure are computed and saved in LCPP repository. The *Imports* area in the LCCP template lists all LCPPs imported by the LCPP being edited (or composed) in the template. Whenever the text in this area is edited, and the changes are recorded, the relationship in the repository is redefined.

An interface specification sets up a link to Larch tier through the **uses** clause. For example, the **uses** clause in RWZone mentions three traits: **Zone**, **Time**, **String**. Moreover, the parameters for **Time**, and **String** are specified. In general, the **uses** clause lists all the LSL traits and the actual parameters to be used for each trait. The *Uses* box in the LCPP template lists the traits used by the LCPP composed (or edited) in the window. The text in this box can be edited: parameters for traits may be refined, traits may be added or removed from the list. At the time of editing this box, the LSL window may be opened to view the traits listed in the box. After completing an editing session the changes made may be committed, thus recording the revised LCPP. Alternately, the status of the box may be left unchanged.

An interface specification may be refined to another interface specification preserving the list of imported specifications and the list of LSL traits used in the specification. For example, a specification F obtained by modifying one or more member functions of an interface specification G is a refinement of G. Another possible way of refinement is to let F include all member functions of G and have new member functions which use the same LSL traits used by G. Both types of refinements lead to a new *version*. It is easy to see that the *version* relationship on the set of all LCPPs is both irreflexive and transitive. The hierarchical dependency of *version* relationship is computed using Warshall's algorithm [10]. A hierarchy can be viewed or traversed. A

```
 typedef int Zone;
 typedef int RWCString;
 imports typedefs;
 struct RWDaylightRule;
 extern Zone local;
 extern Zone standard;
 enum DstRule NoDST, NoAm, WeEu;
 extern RWDaylightRule *rules[3];
 enum StdZone zone;

abstract class RWZone
{
uses Zone, Time(RWBoolean for Bool), string(RWCString for C);
 public:
virtual int timeZoneOffset()
 {
 ensures result = self^.standardOffset;
 }

virtual int altZoneOffset()
 {
 ensures result = self^.DSTOffset;
 }

virtual RWBoolean daylightObserved()
 {
 ensures result = daylightObserved(self^);
 }

virtual RWBoolean isDaylight(const struct tm* tspec)
 {
 requires daylightObserved(self^) ∧ (*tspec)^.tm_wday =
 week_(day(date((*tspec).tm_day, (*tspec)^.tm_month,  (*tspec)^.tm_year)));
 ensures ∃t : Time(result = (t = get((tspec*)^))∧
 observedDST(t.year, self^) <> NON
 ∧(observedDST(t.year, self^) = AHEAD ⇒
 (convert(t) >= convert(beginDST(t.year, self^))∧
 convert(t) <= convert(endDST(t.year, self^))))∧
 (observedDST(t.year, self^) = BEHIND ⇒
 (convert(t) <= convert(beginDST(t.year, self^))∧
 convert(t) >= convert(endDST(t.year, self^)))));
 } ...
```

Figure 31: Larch/C++ Interface Specification for RWZone

Imports List:

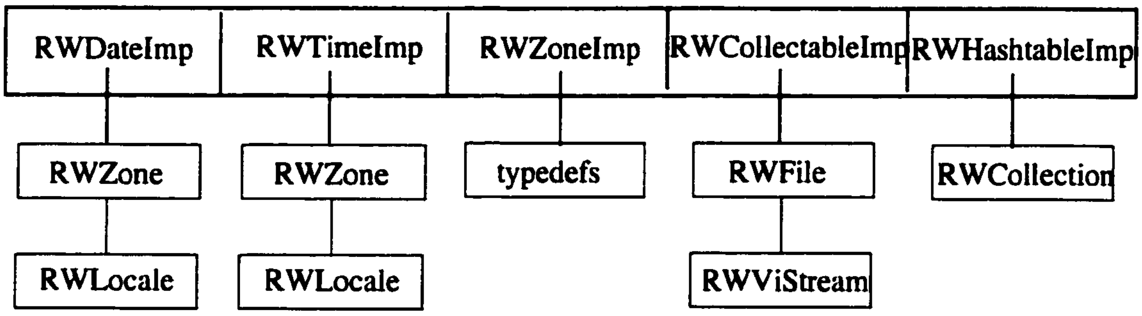


Figure 32: An Example of **imports** List of LCPP Files

version may be added or deleted from a hierarchy by naming the root of the hierarchy and the immediate predecessor of the version; see query structures below.

After completing the editing of the boxes in the LCPP template, click "Generate" to generate the edited specification in the text area of the "Development Window". When a new specification is generated, the system recomputes the relationships between the new LCPP with LSL and other LCPPs. The relationships of the revised LCPP to other components in the reuse repository are recorded by changing the file pointers associated with the LCPP. It is easier to understand the changes by following the links in the structures shown in Figures 32, 33, 34. By clicking on "Revert", the specification is reverted to the text it was when loaded in the template. The template can be closed by clicking on "Close" without recording any of the changes.

A *Query Session Window* as shown in Figure 52 may be created at any time by the user to query the properties of the LCPP specifications in the repository. In this window, there is a radio button corresponding to each query. A query is stated in natural language against the button. By clicking on a button, the user is asking for information against that query. The system opens up a dialog box for the user to input the parameters required to process the query. Upon completing the dialog box information sections, the user must click on "Apply" button to activate the search methods for retrieving the information from the repository. The system response is displayed in a text window. The following sequence of dialogs leads to query selection:

1. If the user chooses `Query on LCPP`, the user is asked to choose one of the following:

    - Query on imports or uses relationship

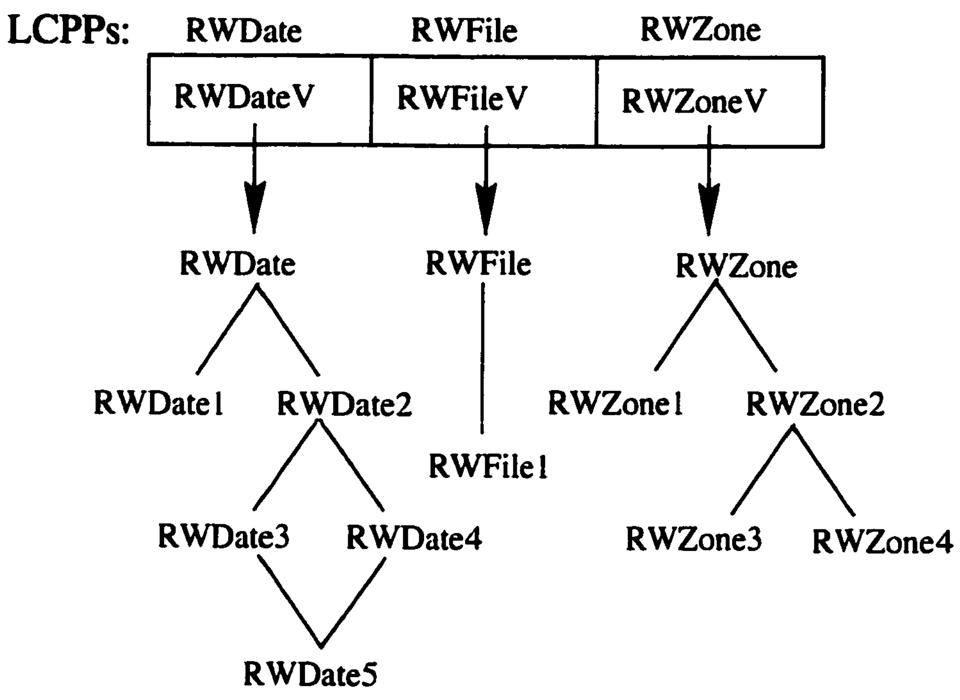


Figure 33: Version List of reusable LCPP files

- Query on version relationship
- Exit
- Previous Menu

2. If `Query on imports or uses relation` is chosen, the user is asked to choose one of the following options:

   - Query on imports relation
   - Query on uses relation
   - General information (Statistics)
   - Exit
   - Previous Menu

3. `Imports, Uses, and Version Relations` - The following queries are answered here:

- **Query-1** Find the transitive closure of the imports relation for a given LCPP Specification.

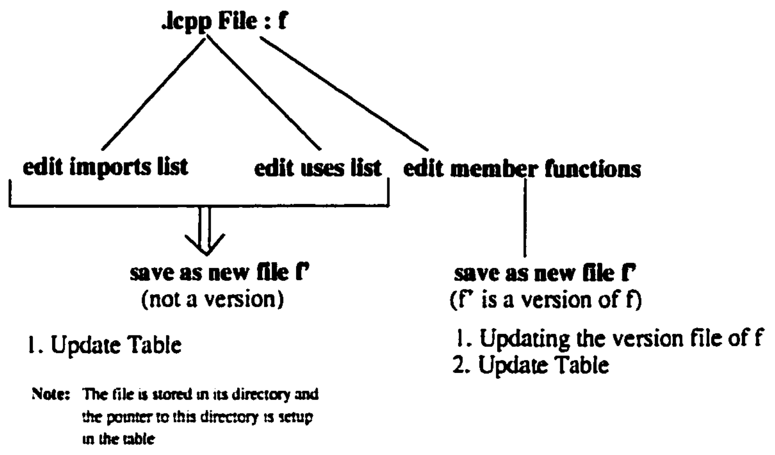


Figure 34: Notion of Save

The member function *Query_1()* of the class **QH_LCPP** processes the above query.

- **Query-2** Find the transitive closure of the uses relation for a given LCPP Specification .

  The member function *Query_2()* of the class **QH_LCPP** processes the above query.

- **Query-3** Find the hierachy representation of the imports or uses or version relation for a given LCPP Specification .

  The member function *Query_3()* of the class **QH_LCPP** processes the above query.

- **Query-4** Find the list and number of LCPP Specifications which uses a given LSL trait.

  The member function *Query_4()* of the class **QH_LCPP** processes the above query.

6.3.1 How queries are answered?

In the current repository we do not have versions of LCPP specifications. So, we restrict to queries on the other two relations.

- **Query-1** Given a LCPP specification A, output all the LCPP specifications imported (used) by A.

The query is answered as follows:
We search the file *Impo_matrix.dat* and display the specifications which have the value 1 corresponding to the specification A. The programming file *Reln_imports.cpp* in the directory "VISTA-C++/Stage_2/imports" answers the above query for the *imports** relation . The programming file *Reln_uses.cpp* in the directory "VISTA-C++/Stage_2/uses" answers the above query for the *uses** relation.

- **Query-2** Given a trait, output all the LCPP specifications which uses A.

  The query is answered as follows:
  We display all the LCPP specifications which have the LSL trait in the *uses* relation. The programming file *used_by.cpp* in the directory "VISTA-C++/Stage_2/uses" answers the above query for the *uses** relation.

- **Query-3** Given a LCPP specification A, find the other LCPP specifications which are imported by A.

  The query is answered as follows:
  We search the file *Imports.dat* for the LCPP specification and return all LCPP specifications following the LCPP specification A.

- **Query-4** Given a LCPP specification A, find the hierachy of the LSL traits that A uses. The query is answered as follows:
  The hierachy algorithm used for the *includes* relation is used. In order to find the first level traits that A uses ,the file *Imports.dat* is used and the hierachy for those LSL traits are obtained by using *InputFile.dat* for all other traits.

6.3.2 Managing TROMs and Subsystems

The template for TROM is identical to its textual structure and is shown in Figure 49. A TROM class can be viewed or edited in this template. The text of LSL traits included in the TROM currently loaded in the template can also be viewed; however, they cannot be edited. In order to edit traits included in the TROM currently viewed in the template, the user should first invoke the *Specification Development Window* for LSL, load the trait to be edited in that window, edit and save it. When the TROM is reloaded in its template the modified trait will be part of its description.

The template for a subsystem is shown in Figure 51. This is identical to the specification template shown in TGC Example. A subsystem can only be viewd in this template. The GUI of TROMLAB need not be invoked for editing or composing a subsystem.

6.3.3 TROM relationships

Two TROMs may be related through one or more of the following properties:

- They share a nonempty subset of traits.
- One of them includes a trait A which is a version of trait B included in the other trait.
- They share a nonempty subset of port types.
- They share the same set of external (input/output) events.

In addition, there may exist relationship at the object level: an object of a TROM class A and an object of a TROM class B are related by the fact that they are both included in a subsystem.

The TROM theory [1] defines three kinds of subtype inheritance relationship, which in turn produce three kinds of versioning for a TROM class. The inheritance properties should be checked by the user; the browser only manages versions. A brief informal basis on which versions may be based is given below; consult [1] for a formal treatment.

Inheritance is a relationship between two classes. Since a class represents a TROM, a subclass defined by inheriting a class also represents a TROM. In other words, inheritance helps to define a TROM using the class definition of another TROM. A formal definition of inheritance in the framework of the TROM model is given below.

Inheritance

Let C' be a class representing a TROM $\mathcal{A}' = (\Sigma', \Theta', \mathcal{X}', \mathcal{T}', \Phi', \Lambda', \Upsilon')$. Let C be a class derived from C' by inheritance. Then the TROM represented by C is defined as $\mathcal{A} = (\Sigma, \Theta, \mathcal{X}, \mathcal{T}, \Phi, \Lambda, \Upsilon)$ such that,

- $\Sigma = \Sigma' \cup \mathsf{Events(C)}$
- $\Theta = \Theta' \cup \mathsf{States(C)}$

- $\mathcal{X} = \mathcal{X}' \dagger$ Attributes(C)
- $\mathcal{T} = \mathcal{T}' \cup$ Traits(C)
- $\Phi_{at} = \Phi'_{at} \dagger$ Att-function(C)
- $\Phi_s = \Phi'_s \dagger$ Hierarchy-Function(C)
- $\Lambda = \Lambda' \dagger$ Trans(C)
- $\Upsilon = \Upsilon' \dagger$ Time-constraints(C)

where $\dagger$ means redefinition.

It follows from the definition that, all the events, states, and traits in the parent class are available in the child class. New events, states, and traits may be added. However, there is no means to block an event, a state, or a trait from being inherited. With regard to the attribute set, new attributes may be introduced or the types of the existing attributes may be modified. Similarly new state hierarchy function or attribute function may be added or the existing functions be modified. Also new transition specifications and time-constraints may be introduced or the existing ones may be modified. GUI provides the facilities for verifying inheritance relationship.

Subtyping ensures some form of behavioral relationship between two types. Hence subtyping is defined in terms of the relationship between the computations of two types. The notion of subtyping is useful for *modular reasoning.* That is, a property verified to be true in a type T will necessarily be true in a subtype of T. Modular reasoning can be useful in the design and verification of concurrent reactive systems since verified subsystems can be put together to build larger systems without reproving already proved proof obligations. For example, checking that an object O_1 is a subtype of object O_2 requires one to make sure that use of object O_1 does not invalidate any assumptions made about the behavior that one could derive from the specification of object O_2. Intuitively, if $\mathcal{A}$ is a subtype of $\mathcal{B}$ then $\mathcal{A}$ preserves all the properties of $\mathcal{B}$. The three kinds of subtyping are:

- *Behavioral-subtyping:* ensures the principle of substitutivity and preserves all the properties of the supertype. An important purpose of behavioral-subtyping is to move from an abstract level to a detailed level by adding more computational behavior.

- *Extensional subtyping:* does not ensure the principle of substitutivity. However, all the properties of the supertype are preserved. Extensional subtyping provides more detail for state-space, differentiating individual states into substates and possibly provides more events enriching the signature and behavior of ports. An important purpose of extension to specialize a common supertype into variants possibly, by introducing additional stimulus-response behavior.

- *Polymorphic subtyping* ensures the substitutivity principle. That is, any object conforming to a polymorphic-subtype can be used whenever an object conforming to the supertype is expected. Polymorphic subtyping usually appears where a supertype is used to characterize the common aspects of several types. In this form of subtyping, a subtype preserves the behavior of its supertype in the environment of the supertype. However, the behavior is not guaranteed to be preserved in other environments. Intuitively, this form of inheritance corresponds to adding more features to an existing model.

6.3.4 Queries on TROMs and subsystems (SCS)

A number of different relationships between subsystems could be defined based on the relationships among the TROMs composing it. The most important one is the *inclusion* relationship; see Train-Gate-Controller example. Other relations include the common TROMs instantiated, and refinements. These are not considered in the present version of TROMLAB development environment.
The following queries are answered here:

- **Query-1** Find the hierachy relationship of the *assumes* relation of the LSL traits that a given TROM includes.

  The member function *Query_1()* of the class **QH_TR_SCS** processes the above query.

- **Query-2** Find the hierachy relationship of the *includes* relation of the LSL traits that a given TROM includes.

  The member function *Query_2()* of the class **QH_TR_SCS** processes the above query.

- **Query-3** Find the hierachy relationship of the *version of* relation of the LSL traits that a given TROM includes.

  The member function *Query_3()* of the class **QH_TR_SCS** processes the above query.

- **Query-4** Find the transitive closure of the *includes* relation for a given SCS.

  The member function *Query_4()* of the class **QH_TR_SCS** processes the above query.

- **Query-5** Find all the objects instantiated by a given SCS.

  The member function *Query_5()* of the class **QH_TR_SCS** processes the above query.

- **Query-6** Find the port links in the given SCS .

  The member function *Query_6()* of the class **QH_TR_SCS** processes the above query.

- **Query-7** Find all the Subsystems which include the given TROM.

  The member function *Query_7()* of the class **QH_TR_SCS** processes the above query.

- **Query-8** Find all the LSL traits included in the objects within the given Subsystem.

  The member function *Query_8()* of the class **QH_TR_SCS** processes the above query.

- **Query-9** Find the TROM name in the *inheritance of* relationship and also the type of the inheritance .

  The member function *Query_9(y)* of the class **QH_TR_SCS** processes the above query.

6.4 How the queries are handled?

VISTA supports the following queries.

- **Query-1** Given a TROM name, find if the TROM exists or not in the database and if it exists give the included traits and for each trait included in the TROM, show the includes hierachy, the assumes hierachy and the version hierachy.

  The query is answered as follows:
  We form the matrix representation for the TROMs and the LSL traitsit includes. We use *InputFile.dat* , which contains the *includes* relation of a given LSL trait to compute the includes hierachy. The *assumes* relation uses *Assumes.dat* and the *version of* relation uses *Version.dat* in order to form the hierachy. The programming file *h_includes.cpp* in the directory "VISTA-C++/Stage_5" gives the includes hierachy. The programming file *h_assumes.cpp* in the directory "VISTA-C++/Stage_ 5" gives the assumes hierachy. The programming file *h_versions.cpp* in the directory "VISTA-C++/Stage_5" gives the versions hierachy.

- **Query-2** Given a subsystem A, determine the transitive closure of the includes relationship for subsystem.

  The query is answered as follows:
  The included subsystems for the given subsystem are extracted from the file *SCS_DATA.dat* and the union of the TROM objects instantiated are extracted from the file *SCS_TROMS.dat* . The programming file *Query_2.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-3** Given a subsystem, determine all the objects instantiated in the subsystem.

  The query is answered as follows:
  The subsystem file is opened and the list of instantiated objects given under the sub-title *Inst:* are displayed. The programming file *Query_3.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-4** Given a subsystem, show all the port links in the subsystem.

  The query is answered as follows:
  The subsystem file is opened and the list of port links given under the sub-title *Config:* are displayed. The programming file *Query_4.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-5** Given a TROM, determine all the subsystems which include this TROM.

  The query is answered as follows:
  The file *SCS_TROMS.dat* is used to form the (0,1) matrix having the subsystems in the rows and the TROMs as columns. The corresponding subsystems which have the value 1 for the given TROM are displayed. The programming file *Query_5.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-6** Given a subsystem, determine all the traits included in the objects within the subsystem and for all the included subsystems.

  The query is answered as follows:
  The list of included subsystems and the instantiated objects of each of the subsystems are determined using the two data files *SCS_DATA.dat* and *SCS_TROMS.dat* and then the list of traits are found using the file *Trom_data.dat* . The programming file *Query_6.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

It is possible to add a TROM file to the TROMs directory and add a subsystem file to the SCS directory.

- **Query-1** Given a TROM file, add the file to the reuse repository.

  The query is answered as follows:
  The TROM file is scanned to get its name, its parameters and the list of traits it includes.

  - Case 1: TROM name exists
    Case 1.1 :List of parameters match
    If the TROM name already exists in the file *Trom_data.dat* and the parameters and LSL name match, then no change to *Trom_data.dat* file is necessary and the already existing file in the TROMs directory is overwritten.
    Case 1.2 : List of parameters do not match
    In case the parameters and the LSL names do not match in the file *Trom_data.dat* we find whether or not the given TROM file is a new version of the already

existing one. If yes, then the version file is updated and the file is saved as a new version in the TROMs directory; otherwise the old file is overwritten to save the new changes to the TROM file.

- Case 2: TROM name does not exist
  If the TROM name does not exist in *Trom_data.dat* then the TROM is added to the *Trom_data.dat* and *TROMS.dat* files and the file is saved in to the directory TROMs.
  In all the above cases, if the new file includes new LSL traits then the files, *TraitFile.dat* and *InputFile.dat* are updated.

The programming file *Add_a_TROM.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-2** Add a given SCS file to add the reuse repository.

  The query is answered as follows:
  The SCS file is scanned to get the SCS name,and the list of objects instantiated by it and the subsystems included by it.

  - Case 1: SCS name exists
    Case 1.1 : List of parameters match
    If the SCS name exists in *SCS.dat* and the parameters , the instantiated objects and the included subsystems match then the old file is overwritten.
    Case 1.2 : List of parameters do not match
    If the SCS name does not exist and the instantiated TROMs are all defined then it is essential to find if the new file is a version file. If yes, the file is saved as a version file and otherwise the old file is overwritten.
  - Case 2: SCS name does not exist
    Case 2.1: List of instantiated objects exist
    If the SCS name does not exist in file *SCS.dat* but the list of TROM names instantiated are present in *TROMS.dat* then the file is saved in the directory SCS and the updates are made in the *SCS.dat* and *SCS_TROMS.dat* files.
    Case 2.2: List of instantiated objects does not exist
    If the SCS name does not exist in file *SCS.dat* and the list of TROM names

do not exist in the *TROMS.dat* then an error message is displayed and the file is not saved.

The programming file *Add_a_SCS.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-3** Given an LSL trait file to add the file to LSL directory.

  The query is answered as follows:
  The LSL file is scanned to get the trait name, its parameters and the list of LSL traits it includes.

  - Case 1: LSL name does exist
    Case 1.1 : List of LSL traits included matches
    If the LSL name already exists in the file *TraitFile.dat* and the list of LSL traits it includes match, then no change to *InputFile.dat* file is necessary and the already existing trait file in the LSL directory is overwritten.
    Case 1.2 : List of LSL traits included do not match
    In case the included LSL names do not match in the file *InputFile.dat* a question is asked to find if the given trait file is a new version of the already existing one. If yes, then the version file is updated and the file is saved as a new version in the LSL directory and the user is questioned if the TROMs that include this LSL trait uses the old version or the new version . The corresponding changes are made in the TROM files. If the given trait file is not a version, then the old file is overwritten to save the new changes to the trait file.
  - Case 2: LSL name does not exist
    If the LSL name does not exist in *TraitFile.dat* then the LSL is added to the *TraitFile.dat* and *InputFile.dat* files and the file is saved in to the directory LSL.

  The programming file *Add_a_LSL.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

- **Query-4** Given a TROM, find the TROMs it inherits from and the nature of inheritance.

The query is answered as follows:
Given the TROM name, we find the nature of its inheritance from the file *inherits.dat* and corresponding to the inheritance, we examine the file *pinherits.dat* or, *einherits.dat* , or *binherits.dat* to find the *version of* relationship.

The programming file *query_version.cpp* in the directory "VISTA-C++/Stage_5" processes the above query.

Chapter 7

The Prototype and Implementation Issues

A prototype reuse search support system to operate within the TROMLAB environment has been built. The reuse repository contains more than 100 LSL traits, 50 Larch/C++ specifications, and about a dozen TROMs and subsystems. There are several TROM classes, and subsystems to be added to the repository. Since the reuse context is specific to TROMLAB, we designed the reuse facility on top of and as a natural extension of system files and directories generated by TROMLAB development components. The repository is fully implemented and tested. It can be run either as part of TROMLAB or a stand alone browser. In the later case, only Larch components can be viewed, composed, and edited.

We have designed and implemented a **Visual Interface for Software Reuse in TROMLAB Applications (VISTA)**, to interact independently with users of Larch components as well as with GUI, the Graphical User Interface of TROMLAB. VISTA is implemented in Sun Microsystems's JAVA/JDK-1.2.

7.1 Riding VISTA

VISTA is a reuse support CASE tool which has two-tiers, namely the visual interface (front-end) and the DB Manager (back-end). VISTA is platform independent as it implemented in JAVA. In this section, we present the visual interface features of the tool and walk-through the user to install and ride VISTA.

7.1.1 Hardware Requirements

VISTA runs on any platform such as Solaris 2.5, Windows 95, Mac OS as long as Sun's Java Virtual Machine(JVM) for JAVA/JDK-1.2 is installed and added to the path.

7.1.2 Software Requirements

We have provided linkage to external software packages from the VISTA to make it a complete software specification development environment and are optional to use the VISTA. These software packages include Larch syntax checkers, lsl, lcc, lp and a C++ compiler (optional).

7.1.3 Functionalities

The Browser environment is very useful for both experienced and new Larch specification developers and/or TROMLAB navigators. The users can easily navigate, query, manipulate and manage the reuse library and versions of the TROMLAB components. This section provides a description of the choices and functionality of each of the VISTA's windows.

7.1.3.1 Getting Started

Before starting the VISTA, the hardware and the software requirements should be met.

To start the VISTA, type *java ObjectBrowser* at the command line on a Solaris or at the MS-DOS prompt on a Windows 95 operating system and the opening window as shown in Figure 35 will appear.

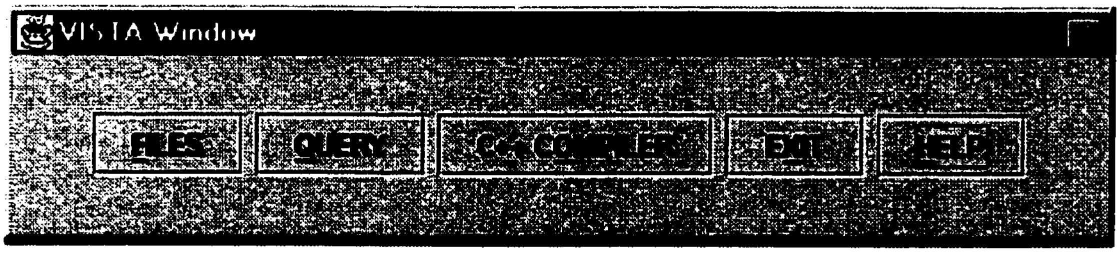


Figure 35: VISTA Opening Window

7.1.3.2 Files

Figure 36 shows the list available TROMLAB components to browse. The TROMLAB components consists of LSL, LCPP, TROM and SUBSYSTEM.

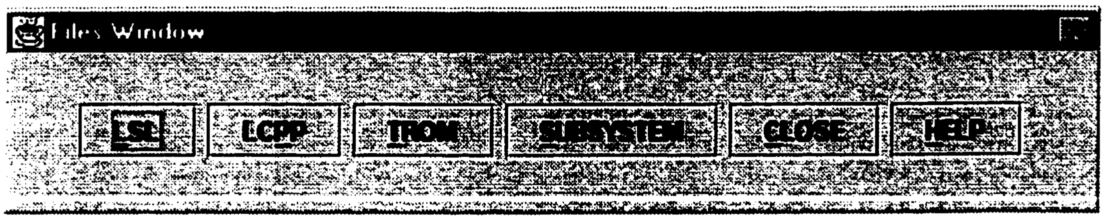


Figure 36: Files Window

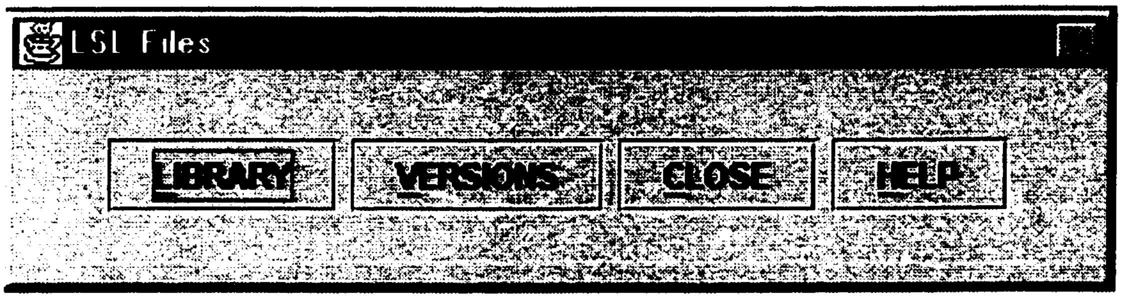
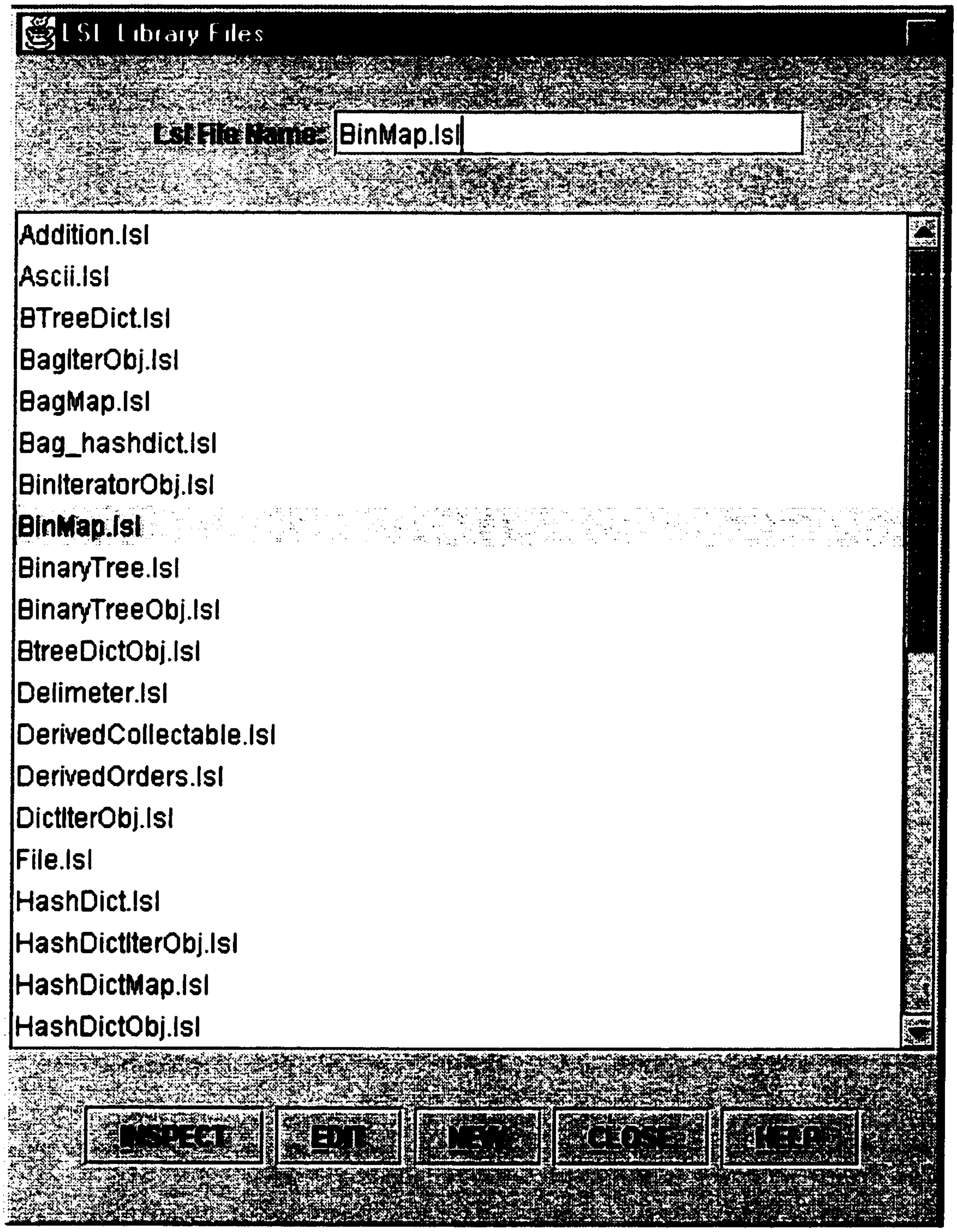


Figure 37: LSL Files Window

7.1.3.3 LSL Files

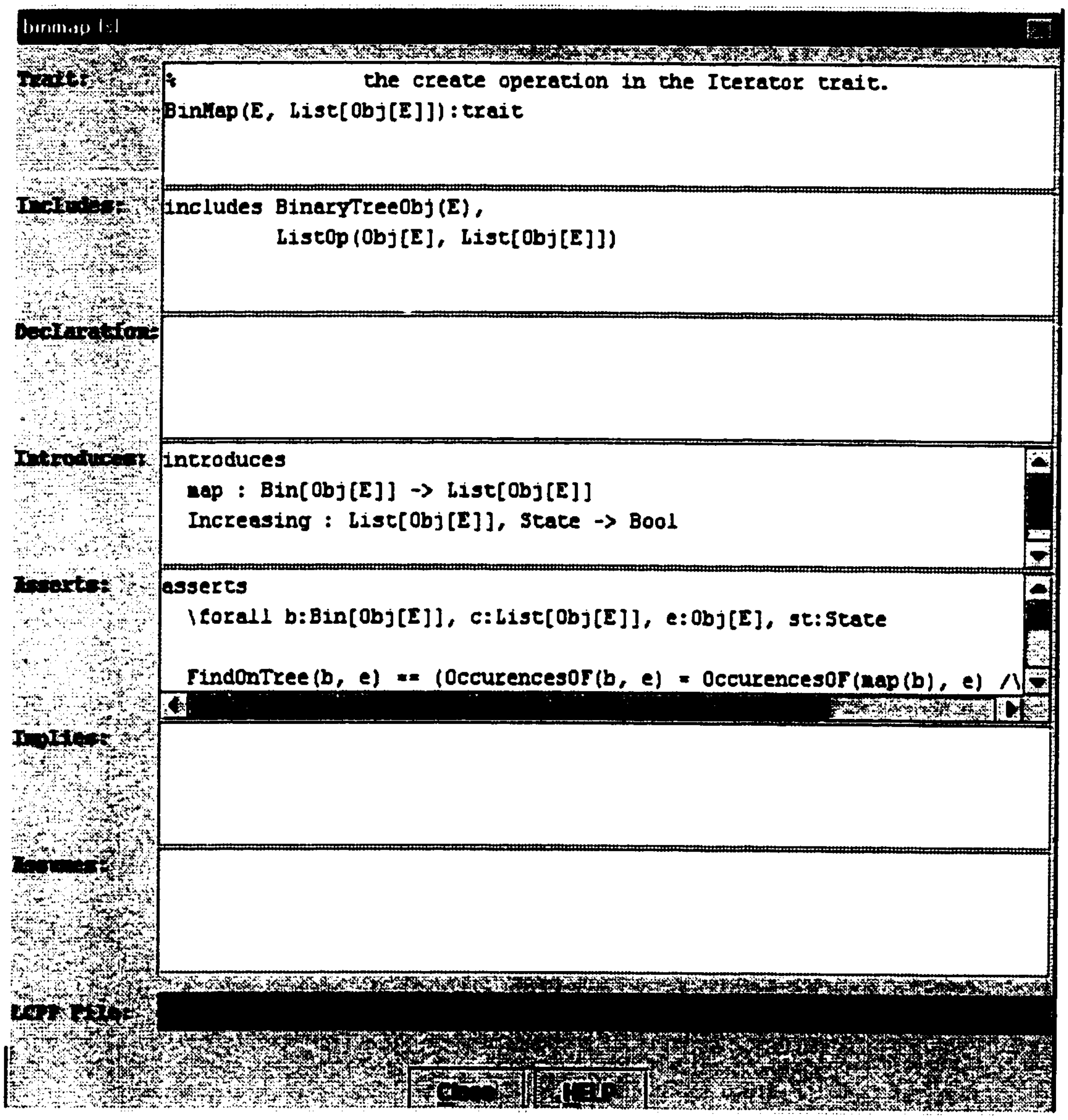
When the LSL button is clicked. the LSL *Files* window appears with 2 choices of browsing the "Library" or "Versions" as shown in Figure 37. Then LSL *Library Files* window appears (Figure 38) if the "Library" is clicked. Similarly, the LSL *Version Files* window appears if the "Versions" is clicked. These windows provides a directory listing and the user can either highlight or type the name of the lsl file and "Inspect" (Figure 39) or "Edit" (Figure 40) it or click "New" to create a new lsl file. The main important point here is that the library files cannot be edited instead a dialog window as shown in Figure 41 pops up and asks the user if he would like to save it as a version file.


LSL Library Files

Lsl File Name: BinMap.lsl

Addition.lsl
Ascii.lsl
BTreeDict.lsl
BagIterObj.lsl
BagMap.lsl
Bag_hashdict.lsl
BinIteratorObj.lsl
BinMap.lsl
BinaryTree.lsl
BinaryTreeObj.lsl
BtreeDictObj.lsl
Delimeter.lsl
DerivedCollectable.lsl
DerivedOrders.lsl
DictIterObj.lsl
File.lsl
HashDict.lsl
HashDictIterObj.lsl
HashDictMap.lsl
HashDictObj.lsl

Figure 38: LSL Library Files Window



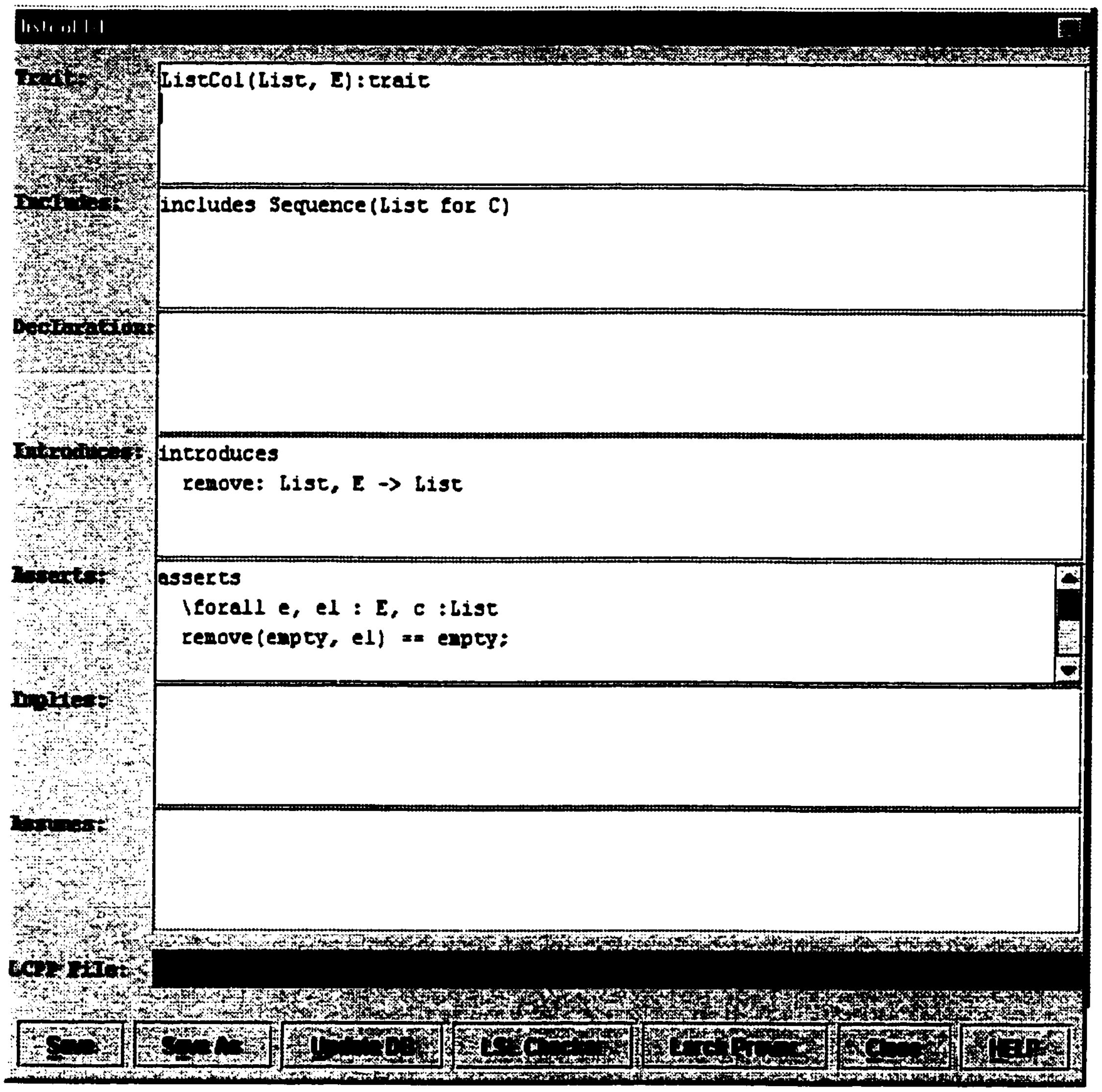
```
%                the create operation in the Iterator trait.
BinMap(E, List[Obj[E]]):trait
```

```
includes BinaryTreeObj(E),
         ListOp(Obj[E], List[Obj[E]])
```

```
introduces
  map : Bin[Obj[E]] -> List[Obj[E]]
  Increasing : List[Obj[E]], State -> Bool
```

```
asserts
  \forall b:Bin[Obj[E]], c:List[Obj[E]], e:Obj[E], st:State

  FindOnTree(b, e) == (OccurencesOF(b, e) = OccurencesOF(map(b), e) /\
```

Figure 39: LSL Inspect File Window



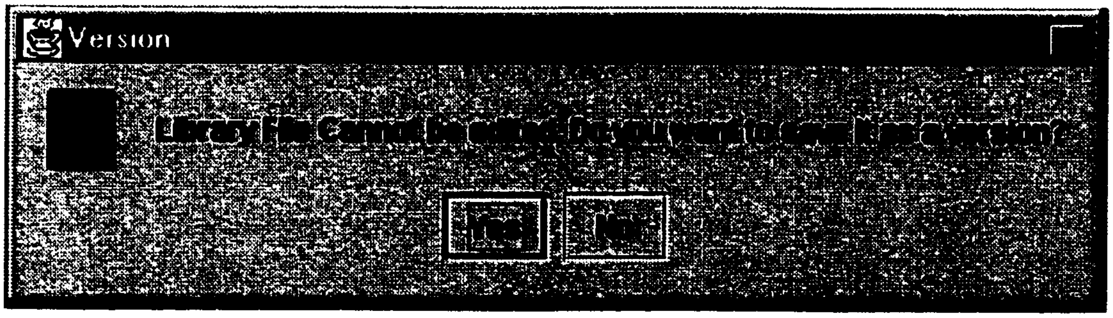
```
ListCol(List, E):trait
```

```
includes Sequence(List for C)
```

```
introduces
  remove: List, E -> List
```

```
asserts
  \forall e, el : E, c :List
  remove(empty, el) == empty;
```

Figure 40: LSL Edit File Window



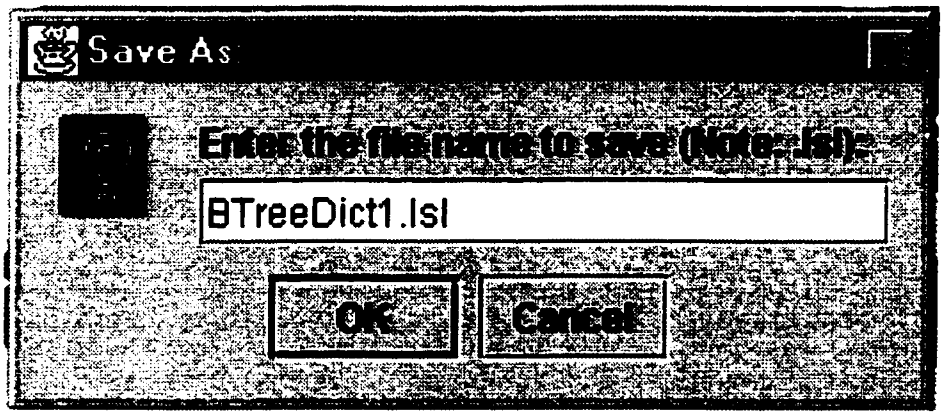
Library File Cannot be edited. Do you want to save it as a version?

Figure 41: Library File Error Dialog Window

- If the user chooses "Inspect", the file is opened in a inspect mode in the LSL template. The corresponding LCPP filename is displayed in the template window in the hypertext mode and user can choose to "Inspect" it as well.

- If the user chooses "Edit", the file is opened in edit mode in the LSL template. Each division of the LSL template is scrollable and allows you to enter the contents in a friendly way. For example, the user can choose enter each of filenames in the *includes* clause seperated by commas in a new line or choose to enter them in a single. The format is preserved even if the file is re-opened. The list of filenames in the *includes* clause are hypertext and user can again choose to either "Inspect" or "Edit" it on the spot. The corresponding LCPP filename is displayed in the template window in the hypertext mode and user can choose to "Edit" it depending on the file opened mode.

7.1.3.4 Saving LSL Files



BTreeDict1.lsl

Figure 42: Save as Dialog Window

In the "Edit" mode, the user has the 3 options to save a file that is opened in the template.

- If the user chooses "Save", the file is saved in the same file name. When a new file is created and if the user tries to "Save", then a "Save as" (Figure 42) dialog pops up and asks for a filename as opposed to a typical save feature in the commercial softwares which will save the file as untitled.

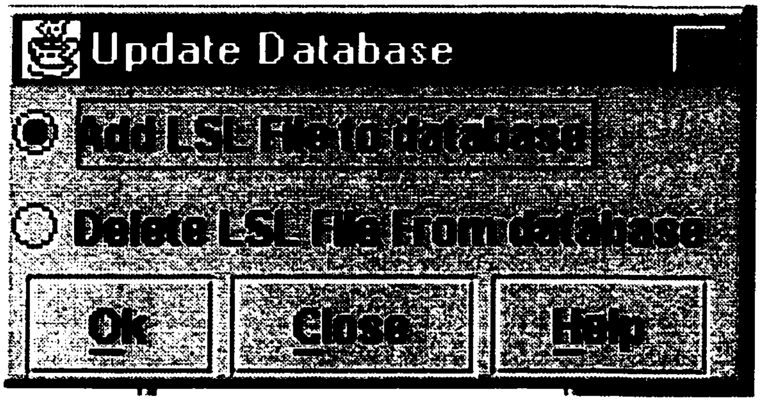


Figure 43: Update Database Window

- If the user chooses "Save As" and if the user enters the same filename as the opened one, a warning dialog pops to confirm the overwriting of an existing file. This button behaves the same way for both old and new files. If the user does not enter any filename, then an error dialog window warns the user.
- If the user chooses "Update DB", then the user can add or delete the LSL trait to or from the version database. The update DB Window is shown in Figure 43. This option is not permitted for the LSL library database. The main purpose of this functionality is to provide version management and answer queries to the user from the database.

7.1.3.5 LCPP Files

When the LCPP button is clicked, the *LCPP Files* window appears with 2 choices of browsing the "Library" or "Versions" as shown in Figure 44. Then *LCPP Library Files* window appears if the "Library" is clicked. Similarly, the *LCPP Version Files* window (Figure 45) appears if the "Versions" is clicked. These windows provides a directory listing and the user can either highlight or type the name of the LCPP file and "Inspect" or "Edit" it or click "New" to create a new LCPP file. The main important point here is that the library files cannot be edited instead a dialog window pops up and asks the user if he would like to save it as a version file.

- If the user chooses "Inspect", the file is opened in a inspect mode in the LCPP template. The corresponding LCPP filename is displayed in the template window in the hypertext mode and user can choose to "Inspect" it as well.

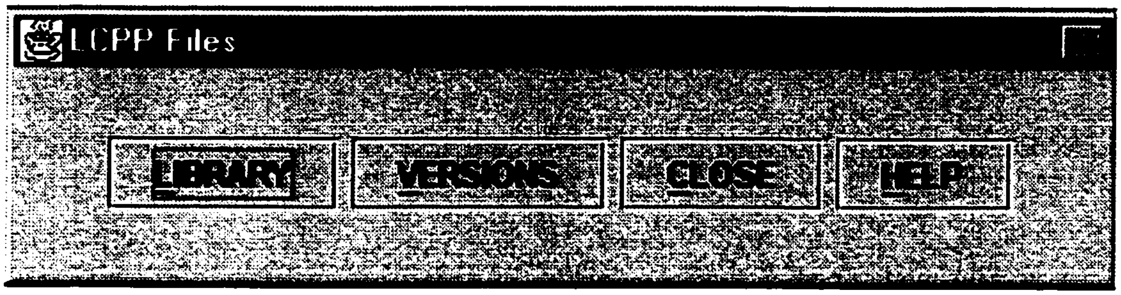
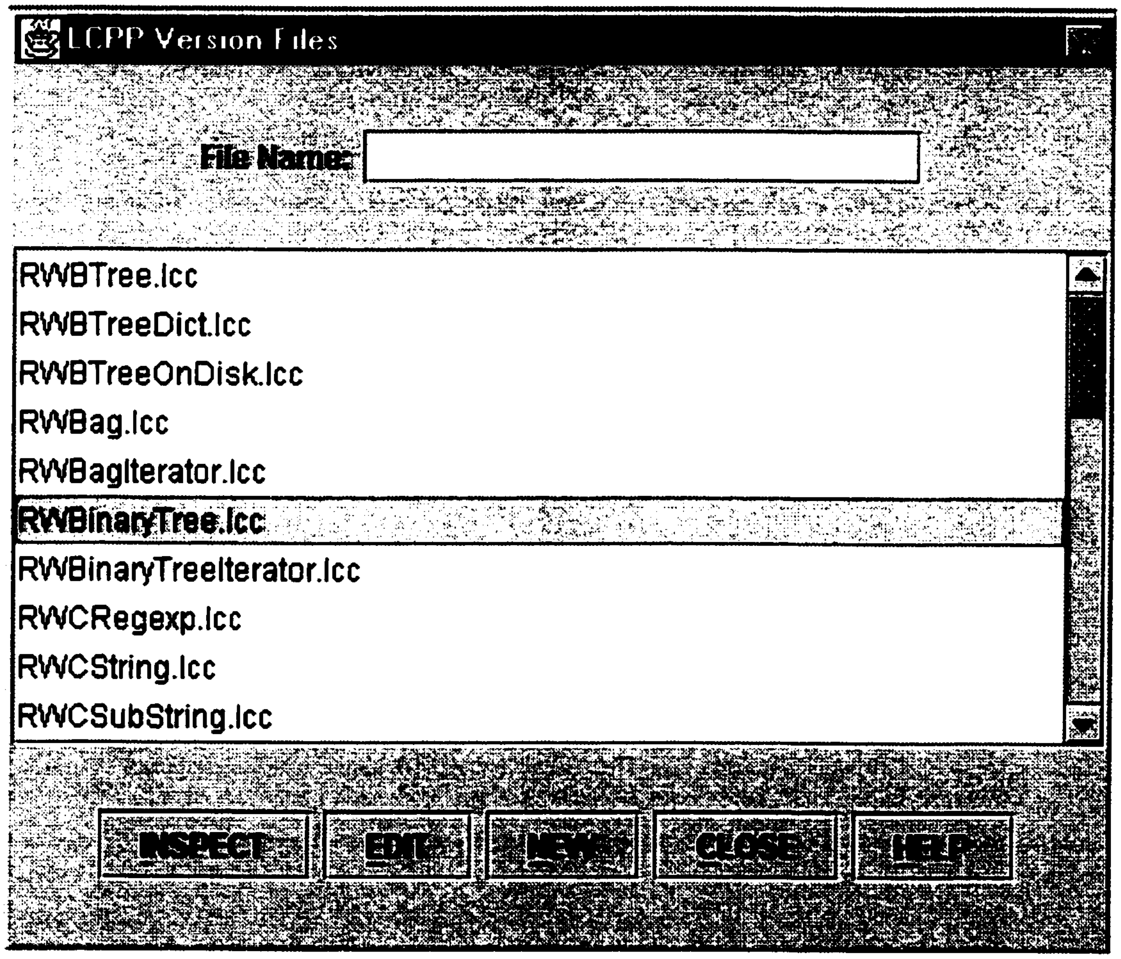
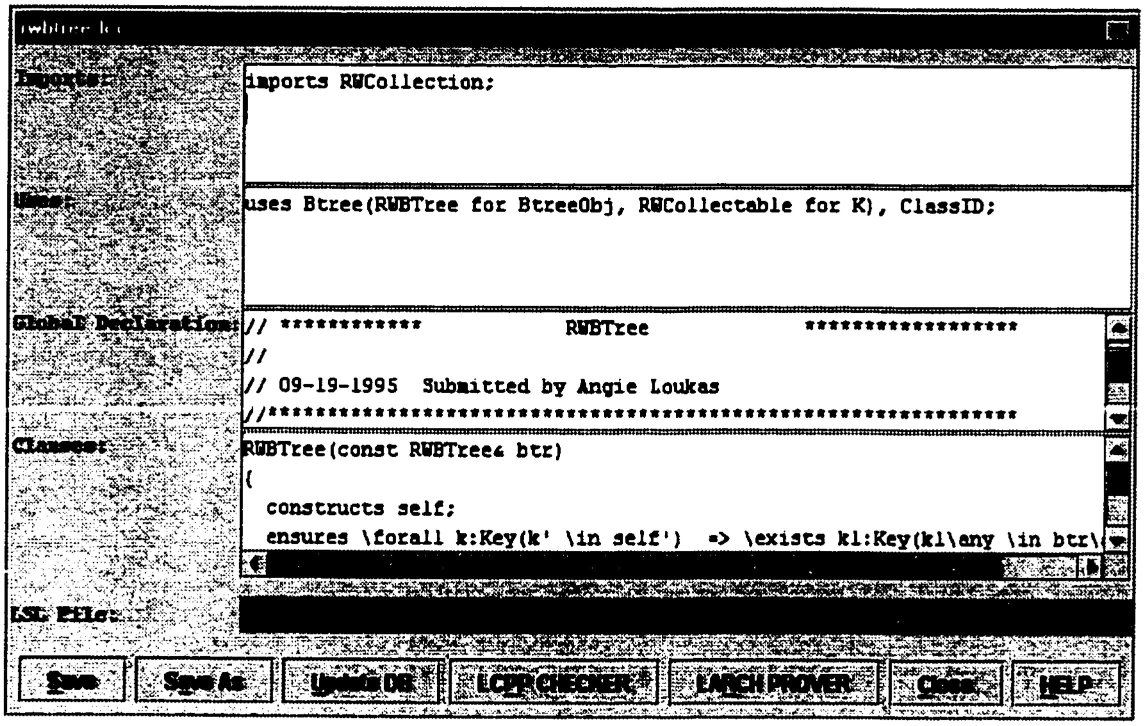


Figure 44: LCPP Files Window



RWBTree.lcc
RWBTreeDict.lcc
RWBTreeOnDisk.lcc
RWBag.lcc
RWBagIterator.lcc
RWBinaryTree.lcc
RWBinaryTreeIterator.lcc
RWCRegexp.lcc
RWCString.lcc
RWCSubString.lcc

Figure 45: LCPP Version Files Window



```
imports RWCollection;
```

```
uses Btree(RWBTree for BtreeObj, RWCollectable for K), ClassID;
```

```
// ************             RWBTree            *******************
//
// 09-19-1995  Submitted by Angie Loukas
//***************************************************************
```

```
RWBTree(const RWBTree& btr)
{
  constructs self;
  ensures \forall k:Key(k' \in self')  => \exists kl:Key(kl\any \in btr\
```

Figure 46: LCPP Editor Window

- If the user chooses "Edit"(Figure 46), the file is opened in edit mode in the LCPP template. Each division of the LCPP template is scrollable and allows you to enter the contents in a friendly way. For example, the user can choose enter each of filenames in the *imports* clause seperated by commas in a new line or choose to enter them in a single. The format is preserved even if the file is re-opened. The list of filenames in the *includes* clause are hypertext and user can again choose to either "Inspect" or "Edit" it on the spot. The corresponding LCPP filename is displayed in the template window in the hypertext mode and user can choose to "Edit" it depending on the file opened mode.

7.1.3.6 Saving LCPP Files

In the "Edit" mode, the user has the 3 options to save a file that is opened in the template.

- If the user chooses "Save", the file is saved in the same file name. When a new file is created and if the user tries to "Save", then a "Save as" (Figure 42) dialog pops up and asks for a filename as opposed to a typical save feature in the commercial softwares which will save the file as untitled.

- If the user chooses "Save As" and if the user enters the same filename as the opened one, a warning dialog pops to confirm the overwriting of an existing file. This button behaves the same way for both old and new files. If the user does not enter any filename, then an error dialog window warns the user.

- If the user chooses "Update DB", then the user can add or delete the LCPP trait to or from the version database. The update DB Window is shown in Figure 43. This option is not permitted for the LCPP library database. The main purpose of this functionality is to provide version management and answer queries to the user from the database.

7.1.3.7 TROMs

When the TROM button is clicked, the TROM *Classes* window appears with 2 choices of browsing the "Library" or "Versions" as shown in Figure 47. Then TROM *Library Classes* window appears (Figure 48) if the "Library" is clicked. Similarly, the TROM *Version Files* window appears if the "Versions" is clicked. These windows provides a directory listing and the user can either highlight or type the name of the TROM class and "Inspect" or "Edit" it or click "New" to create a new TROM class. The main important point here is that the library files cannot be edited instead a dialog window as shown in Figure 41 pops up and asks the user if he would like to save it as a version file.

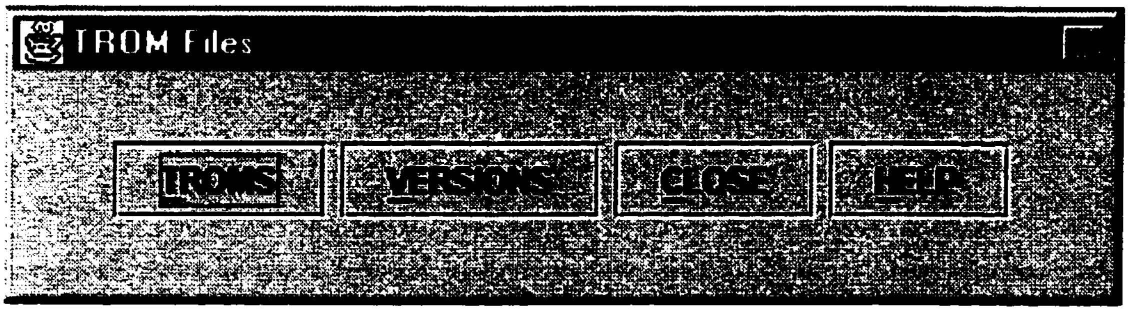
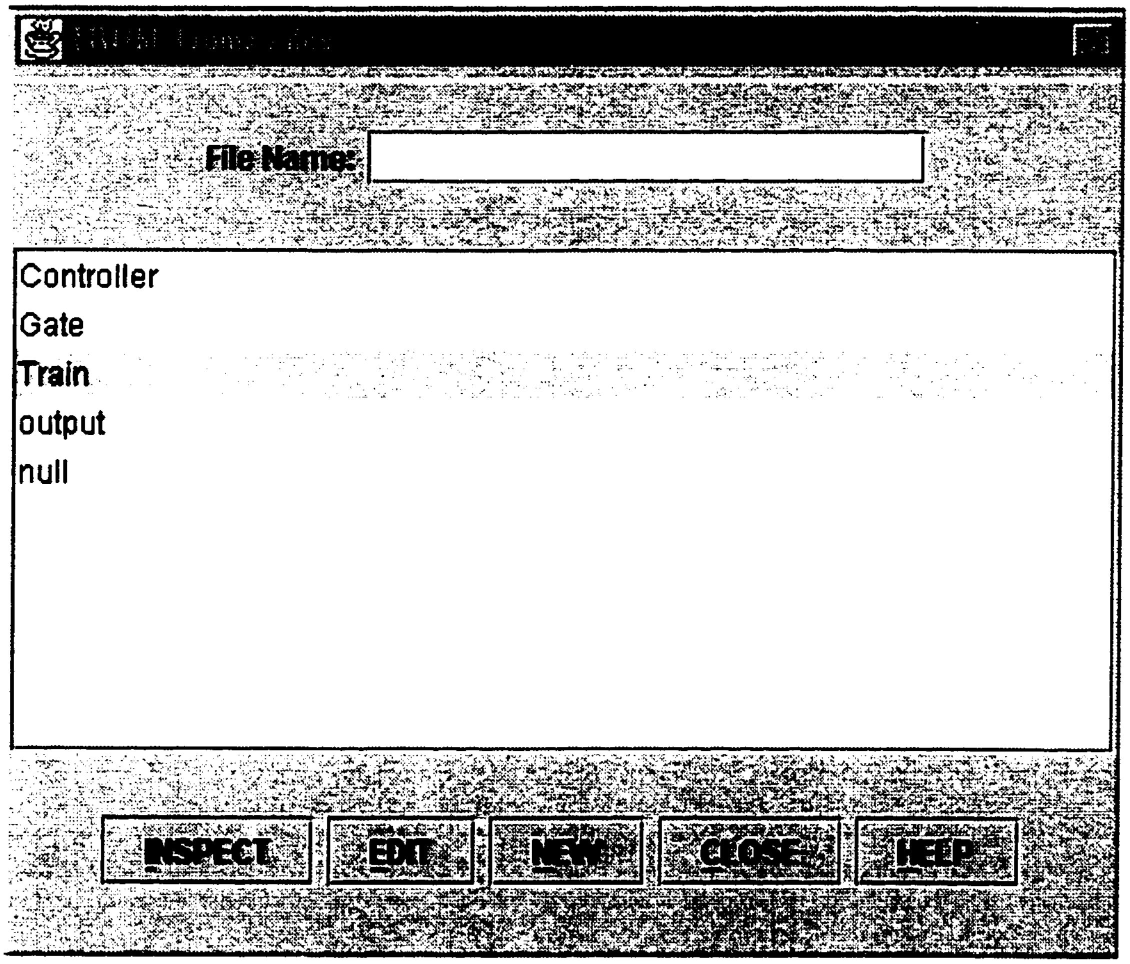
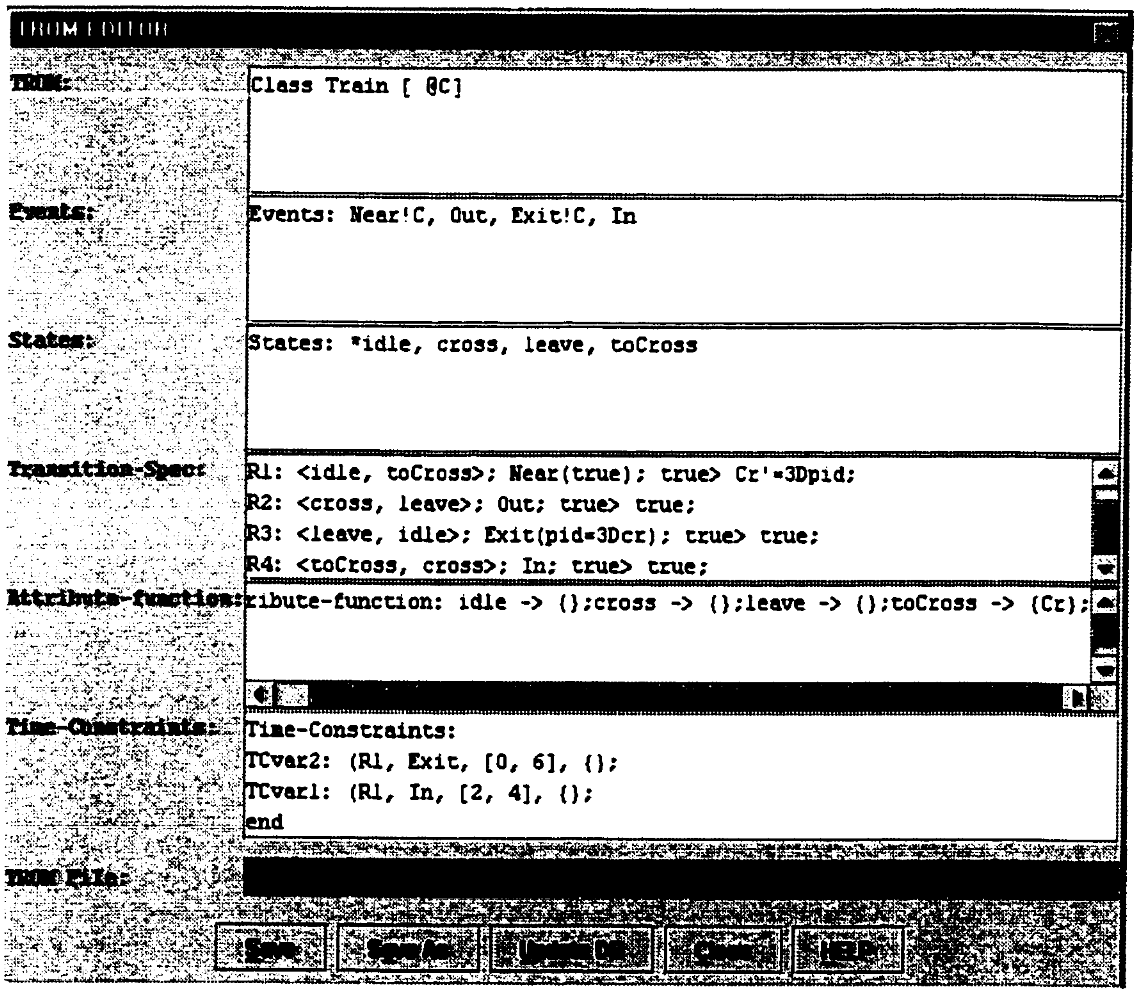


Figure 47: TROM Classes Window



Controller
Gate
Train
output
null

Figure 48: TROM Library Classes Window



```
Class Train [ @C]
```

```
Events: Near!C, Out, Exit!C, In
```

```
States: *idle, cross, leave, toCross
```

```
R1: <idle, toCross>; Near(true); true> Cr'=3Dpid;
R2: <cross, leave>; Out; true> true;
R3: <leave, idle>; Exit(pid=3Dcr); true> true;
R4: <toCross, cross>; In; true> true;
```

```
ribute-function: idle -> ();cross -> ();leave -> ();toCross -> (Cr);
```

```
Time-Constraints:
TCvar2: (R1, Exit, [0, 6], ();
TCvar1: (R1, In, [2, 4], ();
end
```

Figure 49: TROM Editor Window

- If the user chooses "Inspect", the file is opened in a inspect mode in the TROM template. The corresponding TROM filename is displayed in the template window in the hypertext mode and user can choose to "Inspect" it as well.

- If the user chooses "Edit", the file is opened in edit mode in the TROM template (Figure 49). Each division of the TROM template is scrollable and allows you to enter the contents in a friendly way. The format is preserved even if the file is re-opened.

7.1.3.8 Saving TROM Files

In the "Edit" mode, the user has the 3 options to save a file that is opened in the template.

- If the user chooses "Save", the file is saved in the same file name. When a new file is created and if the user tries to "Save", then a "Save as" (Figure 42) dialog pops up and asks for a filename as opposed to a typical save feature in the commercial softwares which will save the file as untitled.

- If the user chooses "Save As" and if the user enters the same filename as the opened one, a warning dialog pops to confirm the overwriting of an existing file. This button behaves the same way for both old and new files. If the user does not enter any filename, then an error dialog window warns the user.

- If the user chooses "Update DB", then the user can add or delete the TROM class to or from the version database. The *Update DB Window* is shown in Figure 43. This option is not permitted for the TROM library database. The main purpose of this functionality is to provide version management and answer queries to the user from the database.

7.1.3.9 Subsystems

When the SCS button is clicked, the *SCS window* appears with 2 choices of browsing the "Library" or "Versions" as shown in Figure 50. Then TROM *Library Classes* window appears if the "Library" is clicked. Similarly, the *SCS Version Files* window appears if the "Versions" is clicked. These windows provides a directory listing and the user can either highlight or type the name of the SCS and "Inspect" or "Edit" it or click "New" to create a new SCS. The main important point here is that the library files cannot be edited instead a dialog window pops up and asks the user if he would like to save it as a version file.

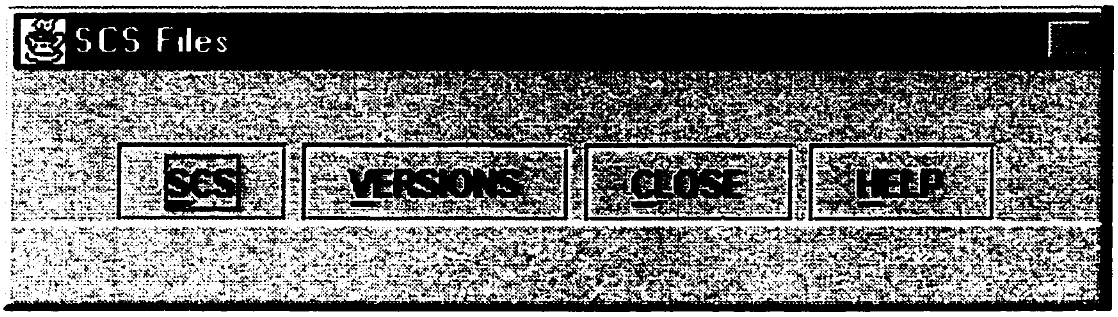


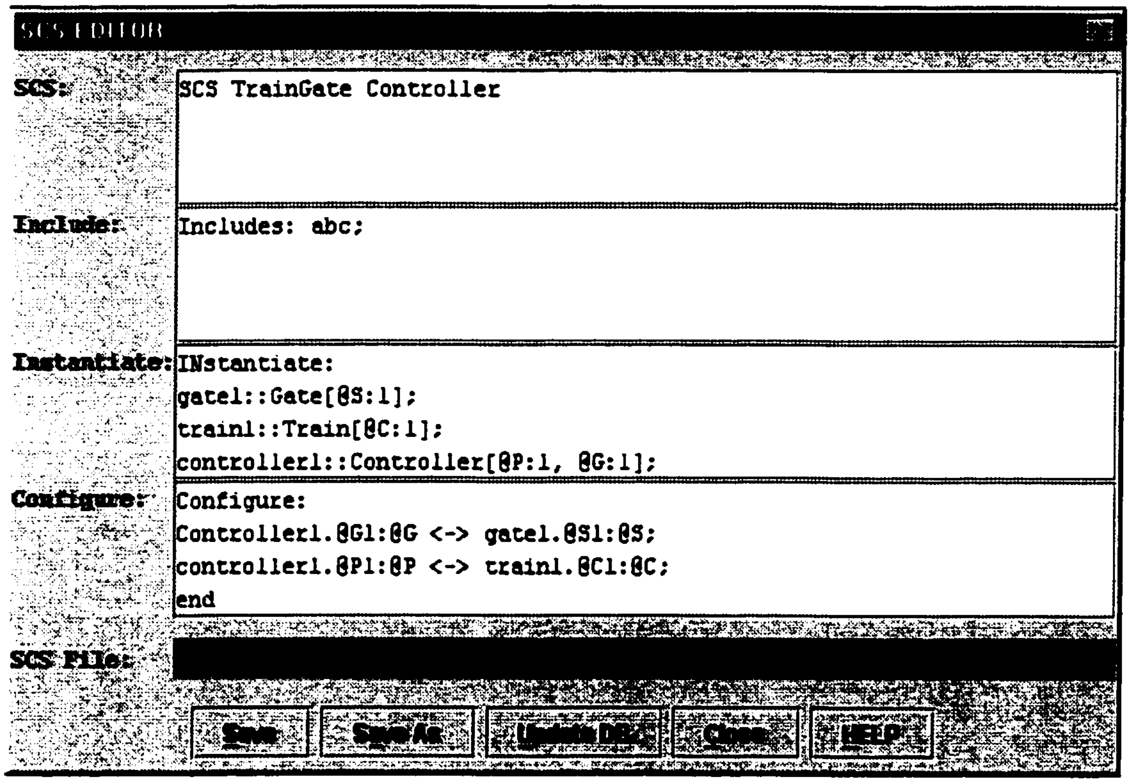
Figure 50: SCS Window

- If the user chooses "Inspect", the file is opened in a inspect mode in the SCS template. The corresponding SCS filename is displayed in the template window in the hypertext mode and user can choose to "Inspect" it as well.
- If the user chooses "Edit", the file is opened in edit mode in the SCS template (Figure 51). Each division of the SCS template is scrollable and allows you to enter the contents in a friendly way. The format is preserved even if the file is re-opened.

7.1.3.10 Saving SCS Files

In the "Edit" mode, the user has the 3 options to save a file that is opened in the template.

- If the user chooses "Save", the file is saved in the same file name. When a new file is created and if the user tries to "Save", then a "Save as" (Figure 42) dialog pops up and asks for a filename as opposed to a typical save feature in the commercial softwares which will save the file as untitled.
- If the user chooses "Save As" and if the user enters the same filename as the opened one, a warning dialog pops to confirm the overwriting of an existing file. This button behaves the same way for both old and new files. If the user does not enter any filename, then an error dialog window warns the user.



```
SCS TrainGate Controller
```

```
Includes: abc;
```

```
INstantiate:
gate1::Gate[@S:1];
train1::Train[@C:1];
controller1::Controller[@P:1, @G:1];
```

```
Configure:
Controller1.@G1:@G <-> gate1.@S1:@S;
controller1.@P1:@P <-> train1.@C1:@C;
end
```

Figure 51: SCS Editor Window

- If the user chooses "Update DB", then the user can add or delete the SCS class to or from the version database. The update DB Window is shown in Figure 43. This option is not permitted for the SCS library database. The main purpose of this functionality is to provide version management and answer queries to the user from the database.

7.1.3.11 Query

The design and algorithm for the processing of different types of queries are described in detail on chapter 6. When the Query button is clicked, the *Query* window appears with 4 choices of query: "Query LSL", "Query LCPP", "Query TROM" and "Query SCS" as shown in Figure 52. Depending on the user's preference, the respective *Query* windows appears as shown in Figures 53, 54, 55, 56. When the user chooses certain search criteria and click "Apply", the query is processed and the results are displayed in the *Query Result(s)* window.

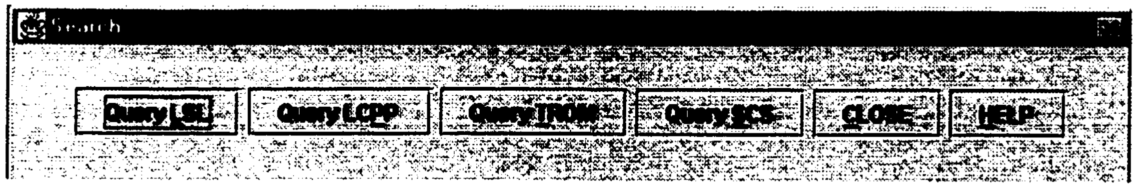


Figure 52: Query Window

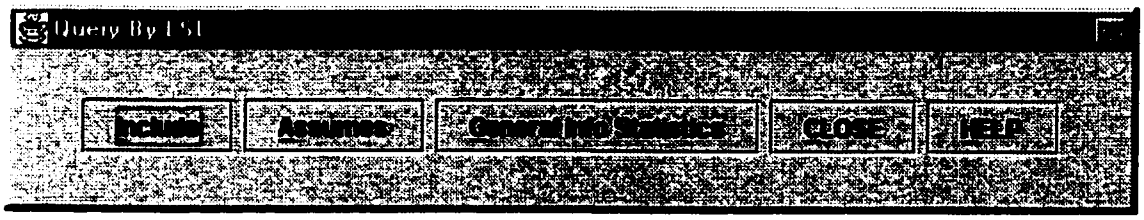
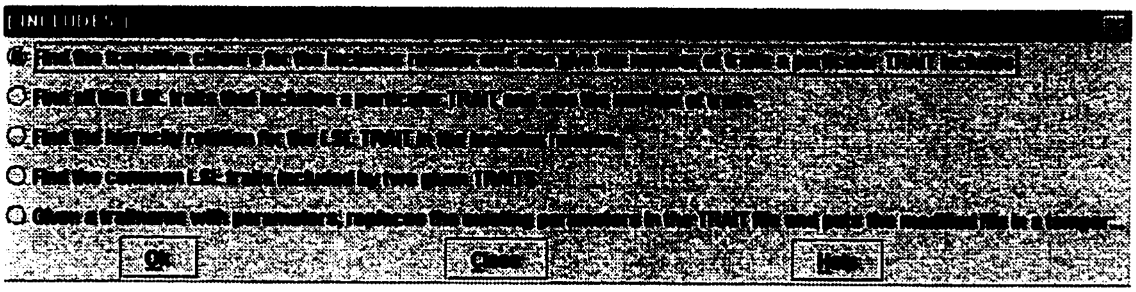


Figure 53: Query LSL Traits Window

7.1.3.12 Exit

When the user chooses to "Exit", the *EXIT* window appears with 2 choices as shown in Figure 57. If the user chooses "Exit VISTA", the VISTA quits or if the user chooses "Exit to TROMLAB", VISTA takes you to the TROMLAB GUI.


INCLUDES

[✓ Find the transitive closure for the includes relation and also give the number of traits a particular TRAIT includes]

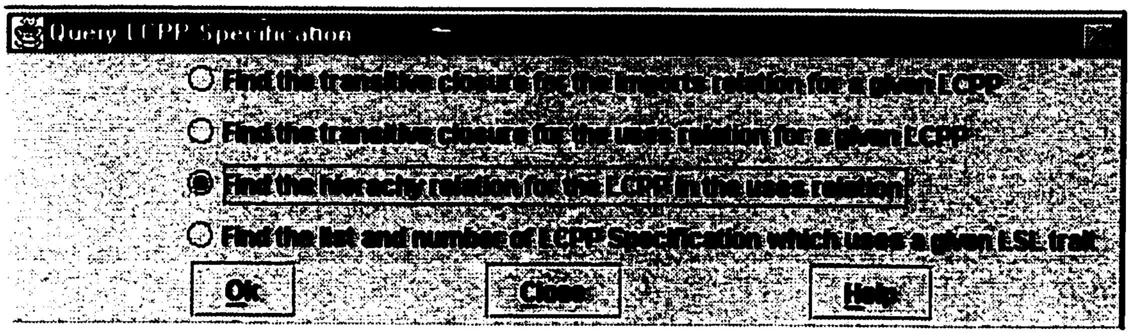
Find all the LSL traits that includes a particular TRAIT and also the number of traits

Find the hieracrhy relation for the LSL TRAIT in the includes relation

Find the common LSL traits included by two given TRAITS

Given a traitname with parameters, replaces the existing parameters in the TRAIT file and puts the modified file in a temporar

Figure 54: Query By Includes for LSL Traits


Query LCPP Specification

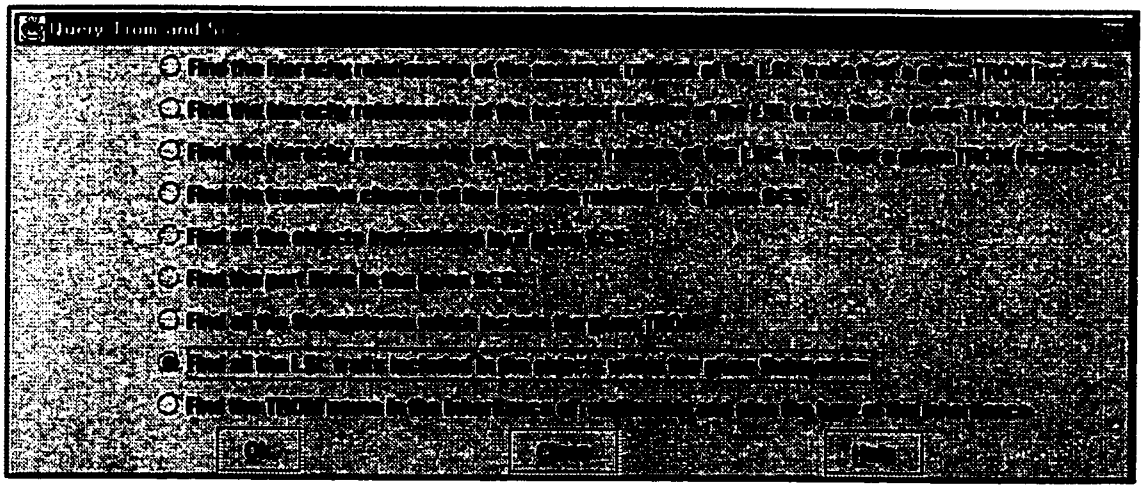
Find the transitive closure for the imports relation for a given LCPP

Find the transitive closure for the uses relation for a given LCPP

[✓ Find the hieracrhy relation for the LCPP in the uses relation]

Find the list and number of LCPP Specification which uses a given LSL trait

Figure 55: Query LCPP Specifications Window


Query Trom and Sc

Find the hierarchy relationship of the [illegible] relation of the LSL traits that a given TROM includes

Find the hierarchy relationship of the [illegible] relation of the LSL traits that a given TROM includes

Find the hierarchy relationship of the [illegible] relation of the LSL traits that a given TROM includes

Find the transitive closure of the [illegible] relation for a given SCS

Find all the objects [illegible] by a given SCS

Find the [illegible] in the given SCS

Find all the [illegible] which [illegible] the given TROM

[✓ Find all the LSL traits [illegible]]

Find the TROM [illegible]

Figure 56: Query TROM Window

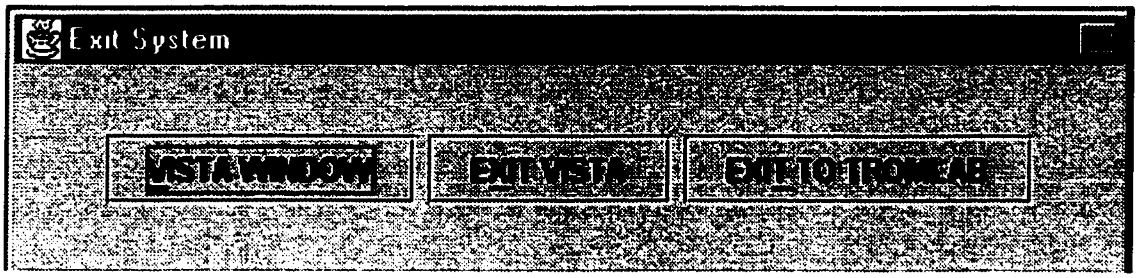


Figure 57: Exit VISTA Window

7.1.3.13 Help

A textual help is provided based on the context of the "Help" button clicked. For example, if the "Help" button is clicked from a LSL *Editor Window*, a sample trait would be provided with textual description of the Larch trait specification. A sample help window is shown in Figure 58.

7.2 Java Classes

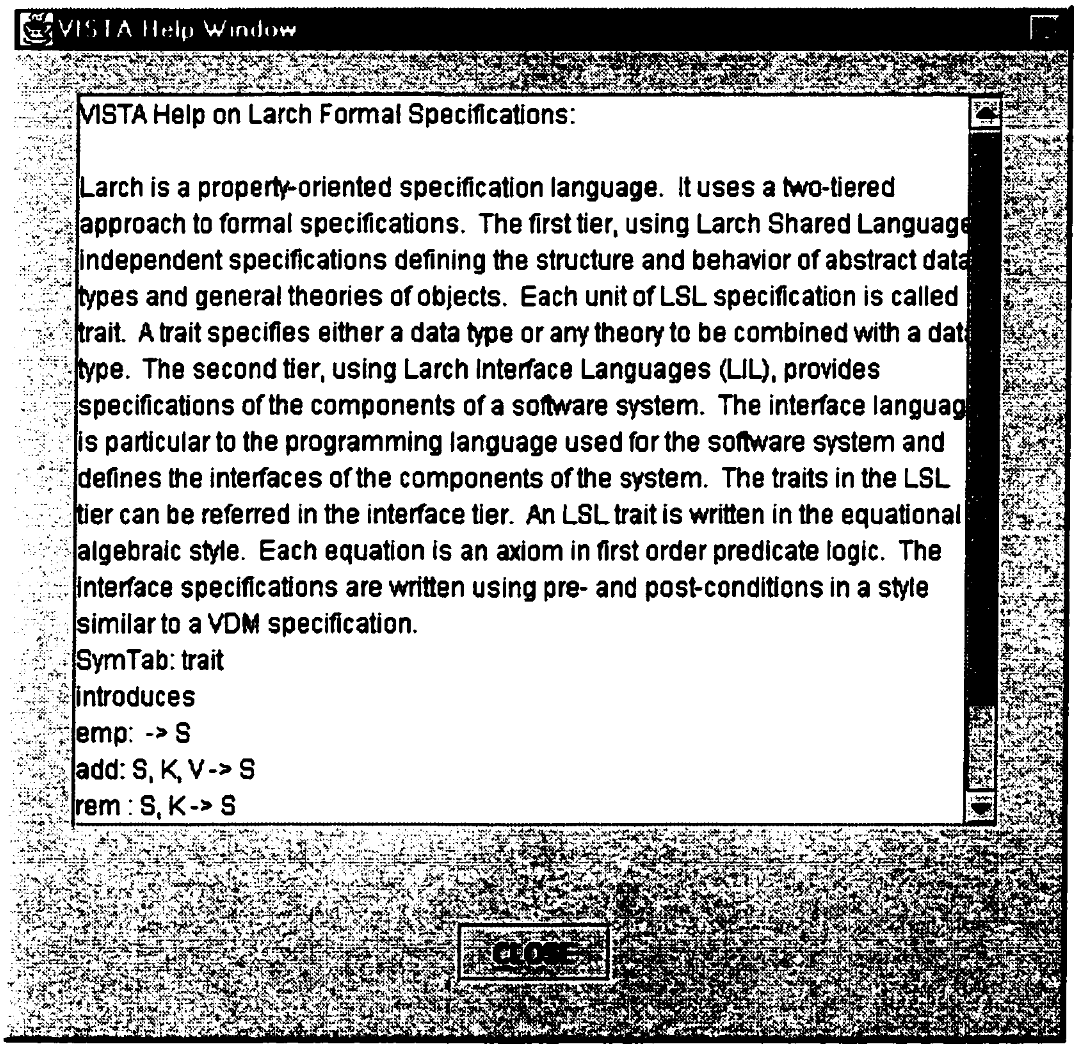
Each of the implemented Java classes of VISTA are designed in such a way that they are independent of each other and can be reused or extended in a another class and modified easily. The java files are located under "VISTA-Java" directory. The documentation and the java class hierarchy created by "javadoc" utility in the "HTML" format are under "VISTA-Java/DOC".

- Vista.java - Main class file for VISTA. It calls "Files", the "Query", "ExitSys" classes and invokes a C++ compiler such as "g++" in a Solaris operating system.

- Files.java - Handles file events for the LSL, LCPP, TROM and Subsystem components

- LSLFiles.java - Manages LSL files under Library and Versions and calls "LSLFilesLib" and "LSLFilesVer" classes

- FileViewer.java - Provides a file inspection viewer super class for all the TROM-LAB components and LCPP files in their respective templates.



VISTA Help on Larch Formal Specifications:

Larch is a property-oriented specification language. It uses a two-tiered approach to formal specifications. The first tier, using Larch Shared Languag independent specifications defining the structure and behavior of abstract dat types and general theories of objects. Each unit of LSL specification is called trait. A trait specifies either a data type or any theory to be combined with a dat type. The second tier, using Larch Interface Languages (LIL), provides specifications of the components of a software system. The interface languag is particular to the programming language used for the software system and defines the interfaces of the components of the system. The traits in the LSL tier can be referred in the interface tier. An LSL trait is written in the equational algebraic style. Each equation is an axiom in first order predicate logic. The interface specifications are written using pre- and post-conditions in a style similar to a VDM specification.

```
SymTab: trait
introduces
emp: -> S
add: S, K, V -> S
rem : S, K -> S
```

Figure 58: Help Window

- LCPPFiles.java - Manages LCPP files under Library and Versions and calls "LCPPFilesLib" and "LCPPFilesVer" classes
- LSLFilesLib.java - Provides dynamic directory listing of library files in a list window with file manipulation operations
- LSLFilesVer.java - Provides dynamic directory listing of version files in a list window with file manipulation operations
- LSLFileViewer.java - Provides the LSL template editor and the hypertext functionality for browsing the hierarchies based on *includes*, *assumes* clauses. It calls "UpDatedb" class to update the relationships of trait files (e.g. includes) library or version files
- LCPPFilesLib.java - Provides dynamic directory listing of library files in a list window with file manipulation operations
- LCPPFilesVer.java - Provides dynamic directory listing of version files in a list window with file manipulation operations
- LCPPFileViewer.java - Provides the LCPP template editor and the hypertext functionality for browsing the hierarchies based on *uses* clause. It calls "UpDatedb" class to update the relationships of LCPP files with trait files and the *uses* (e.g. includes) library or version files
- Query.java - Handle Query events for the LSL, LCPP, TROM and Subsystem components
- QueryLSL.java - Displays and process query based on different search criteria described in Chapter 6.
- QueryLSLInclude.java - Handles query based on Includes relationship in LSL traits
- SCSFiles.java - Manages Subsystem files under SCS and Versions and calls "SCSFilesLib" and "SCSFilesVer" classes
- SCSFilesScs.java - Provides dynamic directory listing of TROMLAB Subsystem files in a list window with file manipulation operations

- SCSFilesVer.java - Provides dynamic directory listing of version files in a list window with file manipulation operations
- SCSFileViewer.java - Provides the SCS template editor and the hypertext functionality for browsing the hierarchies based on included LSL traits or TROM classes
- TROMFiles.java - Manages Subsystem files under TROMs and Versions and calls "TROMFilesLib" and "TROMFilesVer" classes
- TROMFilesTroms.java - Provides dynamic directory listing of TROM classes in a list window with file manipulation operations
- TROMFilesVer.java - Provides dynamic directory listing of version files in a list window with file manipulation operations
- TROMFileViewer.java - Provides the SCS template editor and the hypertext functionality for browsing the hierarchies based on included LSL traits
- UpDatedb.java - Provides an update dialog window to add/delete a trait to the database manager
- ExitSys.java - This class Quits VISTA or takes the user to the TROMLAB GUI.

7.3 Implementation Issues

Two important implementation considerations became evident through the development of the prototype. First, the components in the reuse repository evolve over two time frameworks: (1) during the development of an application; and (2) during the independent development of Larch components for various applications, not necessarily confined to real-time reactive systems. The evolution of components in the first category are handled by the update queries in the system. That is, the developers may invoke these query windows to save new components. Components of the second kind may be added to the system manually. In either case, the relationship among components will be computed after each update. Second, the system must be evaluated from the experiences of people using the TROMLAB environment. For example, slow response times, imprecise retrieval, and incomplete information retrieved will lead developers lose faith in the system. They then would be reluctant to use the search facility. A compromise would be to investigate more appropriate storage schemas and migrate the reuse repository to the new data store. We caution that our browser is not a general purpose reuse engine; it promotes reuse within a very specific context.

Chapter 8

Conclusion

The major motivation for the thesis came from two directions: (1) to provide an on-line facility to view, select and reuse LSL traits and LCPP specifications created by the black-box reuse research (BBRS) group under the guidance of Dr. Alagar [4]; and (2) to provide an active reuse method for TROMLAB components. Since LSL and LCPP are themselves components in TROMLAB, it was decided to build VISTA to achieve both the objectives.

Very little research work has been reported in the reuse of real-time reactive system components. In general, very little reuse is possible with traditional design methodologies. The difficulties are mainly due to the complexity in designs. Modularity of components is an essential aspect for reuse promotion. Designs lacking modularity, compositionality, and specializations are hard to adapt for reuse in assembling large systems. However, these are the foundational features of object-oriented (OO) techniques. Consequently, it becomes essential to integrate OO techniques with real-time requirements. The work on TROM formalism [1] is one of the first attempts in this direction. The advantages of combining real-time with OO, and the challenges it poses in a smooth semantic integration are discussed in [1]. TROMLAB environment promotes this integration in the design and development of real-time reactive systems, and hence the components of TROMLAB are best suited for black-box reuse.

LSL, LCPP, TROM, and SCS components can be reused either individually or collectively in constructing a real-time reactive systems. VISTA provides some limited support for white-box reuse as well. For example, LSL traits from the repository can be reused in either a black-box fashion or can be modified, type checked, and then included in a design. Similarly, a TROM class can be reused in different ways:

- with no change
- developing new classes consistent with the three types of inheritance
- instantiating different objects from it, and using the objects in one or more subsystems

Note that when time constraints are modified, even if changes are certified to be syntactically correct by the *Interpreter*, both simulator and verifier should be invoked to analyze the complete behavior of the system in which it is a component. The TROMLAB GUI should be used for such purposes.

The contributions of this research are:

1. We have brought out the significant advantages of black-box reuse, both in its technical and managerial perspectives.
2. We have proposed a simple approach for the integration of a hypertext-based navigation with a repository of UNIX directories of files for black-box reuse of TROMLAB components.
3. We have shown that repository components can be inspected, edited, composed integrated with a versioning mechanism during the process of reuse.
4. VISTA is implemented in Java and is ready for integration with TROMLAB.

Some of the future research directions are:

- Application of VISTA during the entire Software Life Cycle
- Enhance VISTA to trace dependencies between objects/components at any phase
- Apply heterogeneous database technologies for efficient retrieval and manipulation of objects/components in VISTA
- Create/Integrate an automatic Larch formal specification-based testing tool(s) in VISTA
- Apply this work in a real industry to study the benefits and challenges of Software reuse

Among these the challenging direction of further research is the integration of heterogeneous database technologies with Computer-Aided Software Engineering (CASE) environment for automating the different phases of reuse process. This is the natural step to take when TROMLAB is used by a large community of researchers for the development of large real-time reactive systems.

Bibliography

[1] R. Achuthan, "A Formal Model for Object-Oriented Development of Real-Time Reactive Systems," Ph.D. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1995.

[2] V.S. Alagar, R. Achuthan, D. Muthiayen, "TROMLAB1," *ACM TOSEM* (revised version) September 1998.

[3] V.S. Alagar, R. Achuthan, D. Muthiayen, "Animating Real-Time Reactive Systems," In *Proceedings of Second IEEE International Conference on Engineering Complex Computer Systems*, ICECCS '96, Montreal, Canada, October 1996.

[4] V.S. Alagar, P. Colagrosso, A. Loukas, S. Narayanan, and A. Protopsaltou. "Formal Specifications for Effective Black-Box Reuse", Technical Reports (2 volumes), Department of Computer Science, Concordia University, Montreal, Canada, February 1996.

[5] V.S. Alagar, P. Colagrosso, A. Celer, I. Umansky, R. Achuthan, *Formal Specifications for Effective Black-Box Reuse: Phase I Progress Report*, Department of Computer Science, Concordia University, Montreal, Canada, September 1994.

[6] C.L. Braun, 'Reuse, in Encyclopedia of Software Engineering' J. Marciniak (ed), John Wiley and Sons, 1066–1069 (1994).

[7] A. Celer, *Role of Formal Specifications in Black-Box Testing of Object-Oriented Software*, Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada 1995.

[8] P. S. Chen, R. Hennicker, and M. Jarke, "On the Retrieval of Reusable Software Components," In *Advances in Software Reuse*, IEEE Computer Society Press, Los Alamitos, California, 1993, pp. 99-108.

[9] P. Colagrosso, "Formal Specification of C++ class Interfaces for Software Reuse," M.Comp.Sci. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1993.

[10] T.H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction To Algorithms*, MIT Press, 1991.

[11] W. Frakes and S. Isoda, 'Success Factors for Systematic Reuse' IEEE Software, 15–19 September (1994)

[12] W.B. Frakes, and P.B. Gandel, "Representing Reusable Software", *Inf. Softw. Tech.*, 32, 10, 1990, pp. 653-664.

[13] J.E. Gaffney and R.D. Gruickshank, 'A General Economics Model of Software Reuse' Proceedings of 14th International Conference on Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, 327–337 (1992).

[14] E. Guerrieri, 'Case Study: Digital's Application Generator' IEEE Software, 95–96 September (1994).

[15] J.V. Guttag, and J.J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[16] G. Haidar. "Simulated Reasoning and Debugging of TROMLAB Environment," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, (Expected: Winter 1999).

[17] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M.Politi, R. Sherman, A. Htull-Trauring, M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, April 1990, pp. 403-414.

[18] T. Isakowitz and R. Kauffman, "Supporting Search for Reusable Objects,", *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June 1996.

[19] R. Joos, 'Software Reuse at Motorola' IEEE Software, 42–47, September (1994).

[20] K.R. Koedinger and J.R. Anderson, "Abstract Planing and Perceptual Chunks: Elements of Expertise in Geometry," *Cognitive Science*, Vol. 12, No.4, December 1990. pp. 511-550.

[21] T. Korson and J.D. McGregor, 'Technical Criteria for the Specification and Evaluation of Object-Oriented Libraries' Software Engineering Journal, March (1992).

[22] G.T. Leavens, *Larch/C++ Reference Manual, Draft: Revision 5.1*, February 1997.

[23] W.C. Lim, 'Effects of Reuse on Quality, Productivity and Economics' IEEE Software, **11**, 23–30 (1994).

[24] G. Leavens, Y. Cheon, *A Quick Overview of Larch/C++*, TR No. 93-18, Department of Computer Science, Iowa State University, June (1993).

[25] G.T.Leavens and Y.Cheon, "Preliminary Design of Larch/C++," in U.Martin and J.Wing (Eds.), *Proceedings of the First International Workshop on Larch*, Workshops in Computer Science Series, Springer-Verlag, 1992.

[26] M.D. McIlroy, 'Mass Produced Software Components, In Software Engineering: Report on a Conference by the NATO Science Committee (Garmish, Germany)' P. Naur, D. Randell, EAS, NATO Scientific Affairs Division, 138–150, Brussels, Belgium (1968).

[27] L. Mikusiak, V. Vojtek, J. Hasaralejko, J. Hanselova, "Z- Browser - Tool for Visualisation of Z Specifications," *Technical Report*, Department of Computer Science and Software Engineering.

[28] I.E. Moser, Y.S. Ramakrishna, G. Kutty, P.M. Melliar-smith, and L.K. Dillon. "A Graphical Environment for the Design of Concurrent Real-Time Systems," *ACM TOSEM*, Vol. 6, No. 1, January 1997, pp. 31-79.

[29] M.J. Morin, B.H.C. Cheng, "User Manual for Larch Development Environment," *Technical Report*, Department of Computer Science, Michigan State University, East Lansing, MI 48824, 1994.

[30] D. Muthiayen, "Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1996.

[31] S. Narayanan, "Formal Methods For Reuse Of Design Patterns And Micro-Architectures," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1996.

[32] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," *Proceedings of 11th International Conference on Automated Deduction*, CADE 1992, Vol. 607 of Lecture Notes in Artificial Intelligence, Springer-Verlag, , pp. 748-752.

[33] R. Prieto-Diaz, "Implementing Faceted Classification for Software Reuse," *Communications of ACM*, Vol. 34, No. 6, May 1991, pp. 89-97.

[34] *UML Notation Guide*, Version 1.1, Rational Software Corporation, September 1997.

[35] *UML Semantics*, Version 1.1, Rational Software Corporation, September 1997.

[36] Rogue Wave, *Tools.h++ Class Library*, Version 6.0, Rogue Wave Software, 1993.

[37] G. Salton and M. G. McGill, *Introduction to Modern Information Retrieval*, McGraw Hill, New York, 1983.

[38] B. Selic, G. Gulleckson, and P. T. Ward, *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[39] R. Sessions, 'The System Object Model (SOM): A Technology for Language Independent Objects' Tutorial Notes, OOPSLA (1993).

[40] V. Srinivasan, "An Intelligent Graphical User Interface System for TROM-LAB," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, (Expected: Winter 1999).

[41] A. Tao, "Static Analyzer: A Design Tool for TROM," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1996.

[42] I. Umansky, "Completeness of Larch/C++ Specifications for Black-Box Reuse," Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 1995.

[43] J. Wing, "A Two-Tiered Approach for Specifying Programs," Technical Report TR_299, Massachussets Institute of Technology, Laboratory for Computer Science, 1983.

[44] J. Wing, "Writing Larch Interface Language Specifications," *ACM Transactions on Programming Languages and Systems*, Vol.9, No.1, January 1987, pp. 1-24.

[45] J. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, Vol.23, No.9, September 1990, pp. 8-24.