COMPLETENESS OF LARCH/C++ SPECIFICATIONS FOR BLACK-BOX REUSE

ILYA UMANSKY

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

MARCH 1995

ISBN 0-612-05141-2

Abstract

Completeness of Larch/C++ specifications for black-box reuse

Ilya Umansky

The object-oriented paradigm introduced new capabilities for an effective software development. One of the most promising benefits is the possibility to reuse software components which were built and tested thoroughly. Software reuse is most effective when it is conducted in a black-box fashion. That is, when the software can be used without studying its source code or running tests to clarify its behavior. To achieve a black-box code reuse the reusable software has to be complemented with its specification. Informal description is not adequate for this purpose because it lacks precision and does not have sufficient expressive power. As a result, researchers turn their attention toward formal specifications.

In this thesis we study completeness of the specifications for C++ classes intended for a black-box reuse. We present a definition of an interface behavior of a C++ class and completeness criteria for specifications to be able to convey this behavior. To provide practical means for the completeness verification we identify the completeness verification methodology and present the algorithm to apply this methodology.

The completeness verification algorithm developed in this thesis uses Larch Prover, an automatic proof assistant. We provide guidelines for using Larch Theorem prover when applying the completeness verification algorithm, and provide the means for incompleteness detection and localization as well as incompleteness correction. Finally, we generalize the completeness verification algorithm for C++ constructs having inheritance and virtual functions, and identify the cases when incompleteness can not be removed.

Acknowledgments

I would like to express my deepest gratitude to my supervisor Dr. V.S. Alagar. His guidance, academic wisdom and encouragement made my thesis work a pleasant and extremely educational experience.

I would also like to thank the "Software Reuse team", Piero Colagrosso, Ramesh Achuthan, and most of all, my good friend Alicja Celer. Never before have I had the opportunity to work with a better group of people. The success of the "Software Reuse team" and this thesis would not have been possible without them.

More than anyone else, I would like to thank my wife Irina and my children Sophia, Daniel and Mitchell. It was their continued support and encouragement that made this degree possible.

Contents

List of Figures

Chapter 1

Introduction

Many of the problems in software development have been recognized since the late 1960s. By this time software evolved from stand-alone computational programs into complex systems caring for a diversity of requirements. As those systems grew they became too large for a single person to understand. Software engineering responded with decomposition techniques that let each programmer(or programming team) work in his own corner of the world. The system view became ever more elusive, the internal interface of software systems became increasingly complex, and software became more difficult to reshape and extend. This complexity precipitated a software crisis that has been with us for 20 or 30 years.

1.1 Object-oriented concepts for software constructs

The object paradigm is a decomposition technique that was introduced in the late 1970s. It raised expectations for software quality and productivity among the software engineering communities.

Objects emphasize independence. An object is an island of administration and maintenance, it is an abstraction having features of a real world entity and as such exhibiting a certain encapsulated behavior. Each object is a participant in loosely coupled community of parts. These parts communicate by sending each other messages. The messages result in the invocation of methods (member-functions in the C++

object-oriented language terminology[Strous94])which perform the necessary actions. The sender of the message does not need to know how the message is processed internally by the object, only that it responds to particular message in a well-defined way. Thus, from the point of view of an object's clients only the object's interface behavior is important.

Objects are grouped into classes when they share the same interface; that is, they respond to the same message in the same ways. This allows many objects to be described by only a few classes. Thus, classes are the object's building blocks.

1.2 Black-box software reuse

The concept of software reuse was introduced in a seminal paper by Mellroy [Mc68] at the 1968 NATO Conference. Software developers realized that there is a lot in common between different pieces of software units: similar tasks to perform, similar structures and architectures, similar algorithms and mechanisms. Instead of writing a software component which provides a well-defined functionality from scratch, it can be reused since the component has been written and tested once.

The most effective approach in software reuse is reuse in a black-box fashion. Black-box reuse involves the use of a component without knowledge about its implementation. Though the concept of software reuse has been identified a while ago, it could not be employed without support of the entire software development cycle. Indeed, the level of granularity of the reusable blocks must be identified; commonality of the components must be encapsulated into the reusable units; specification of the unit must be provided to convey to a reuser the knowledge of the unit's features in order to reuse it without going into the unit's code.

Object-oriented paradigm provides support for the first two items identified above. A class, as an object building block, serves as a reusable component comprising of objects instantiated from it and subclasses derived from it. Only a class interface is needed to be known while implementation details can be hidden. This is enabled by encapsulation of the class data and providing interface functions to exercise the class behavior. Software developers succeeded in creating reusable class libraries

which provide building.blocks with generic features that can be used in a variety of software.

1.3 Formal specifications and black-box reuse

As was mentioned above, a specification of a class behavior becomes extremely important to enable black-box reuse. Indeed, a reuser has to know precisely an interface behavior of the class in order to reuse it. Most of the object-oriented languages and methodologies provide little if any to support this aspect.

Usually a verbal description is supplied along with the interface functions. This is supposed to deliver all the information which may be required in order to reuse the class. A great deal of inadequacy exists between an informal description and the complexity of a class behavior to convey this description. Consequently, the programmers have to turn to the implementation of a class to fully understand its behavior, or make improper use of the class which will result in a defective program. As a result, a black-box reuse has not yet succeeded in becoming a well accepted practice for software developers.

Formal specifications, which are based on mathematical abstractions and have precise semantics, are required for a precise description of the functionalities, properties and interface of software components so that potential users can learn exactly how each component is expected to behave in order to write correct programs using the component. Thus, using formal specification is a potential solution to remedy the problem identified for informal descriptions to promote object-oriented software reuse.

One of the most important properties to be satisfied by a formal specification in order to enable black-box reuse is completeness. Indeed, in order to correctly reuse a software component, it is also required to know a complete description of the externally observable behavior of the software component. The precision afforded by formal specifications makes it possible to write succinct interface specifications which are complete.

Theoretically, formal semantics of formal specifications makes it possible to verify

completeness of formal specifications in a semantic manner. Indeed, since formal specifications are based on logic it could be possible to prove their completeness or disprove it. On the other hand, no feasible methodology exists to realize such verification in practice.

This thesis addresses only one part of a larger and on-going research on black box reuse of object-oriented software: algorithmic verification of specifications written in Larch/C++. The foundational work on completeness criteria for Larch/C++ specification was done by Piero Colagrosso [Col93] and this thesis provides an algorithmic solution, demonstrates its effectiveness using Larch Prover(LP) and gives directions on a high level design for building a tool based on the algorithm.

More specifically, based upon the thesis work of Piero Collagrosso[Col93], in this thesis we develop a methodology for completeness verification of formal specifications, written in Larch/C++ interface specification language. The completeness criteria introduced in [Col93] will be developed into the completeness verification methodology and later, into the completeness verification algorithm. As a result, we will provide algorithmic steps to guide verification of the completeness of formal specifications, enabling semi-automation of this process with the help of an automated theorem prover.

This research on black-box reuse is sponsored by BNR-NSERC under a Collaborative Research and Development Grant to Dr. V.S. Alagar. There are a number of students working on different aspects: for example, Alicja Celer, with whom I collaborated to develop Larch/C++ specifications of Rogue Wave class libraries [rogue93], has just completed her thesis devoted to another part of the project (role of formal specifications in black-box testing of object-oriented software); there are four other students who are writing specifications of those classes from [rogue93] not included in [ACCUA94]. They plan to extend the work of this thesis in several directions and build tools based on the collective work of this group to make software reuse feasible in the context of commercially used C++ class libraries. Thus, my thesis is just a small contribution to the overall goals; yet, the results of this thesis are quite significant: this is the first attempt to create an algorithm for completeness verification of Larch/C++ specifications. It is my hope and wish that my results will be used and

tested by our research group who are still developing Larch/C++ specifications for Rogue Wave Library.

In the second chapter we survey the concept of completeness, looking for a formal answer to this question and present a notion of completeness which makes specification to be adequate for a black-box reuse as stated in [Col93]. In the third chapter we will identify a general methodology to ensure that the completeness criteria is satisfied. Fourth chapter presents an algorithm to realize this methodology. In the fifth chapter we will introduce Larch Theorem Prover, an automatic proof assistant, and give an example of completeness verification for a class from commercial C++ library. We will show how incompleteness is detected and suggest a way to remedy and correct incompleteness. Finally, we will extend the approach to more general cases, where virtual functions, inheritance and exception handling are presented.

Chapter 2

Completeness of Formal Specifications

The reuse of a class in a black-box fashion can be promoted only if a reuser has sufficiently complete knowledge of the class behavior. Since our intention is to use formal specification to support a class reuse, it has to be ensured that the specification provides all the necessary information for this purpose. In other words, the specification has to be complete with respect to some suitable completeness criteria .

In this chapter, we first survey different criteria of completeness of formal specifications. Next, the definition of a complete specification for a class intended for a black-box reuse will be presented. Since this thesis concerns with C++ classes specified using Larch/C++ specification language, the chapter includes a brief overview of Larch/C++.

2.1 Survey of completeness criteria

Software development process may employ formalizations for a variety of tasks. The most straightforward one is to formalize the informal requirements of the user. Formalized document will serve as a document for further system development. During system verification, the implementation obtained has to be matched against the specification.

Informally, completeness of a specification as a formalization of the user's informal requirements can be defined as follows[AK94]:
A specification is considered to be complete if and only if its consequence closure is equivalent to that of the set of the client's intended requirements. That is, the specification should contain the same behaviors (or properties) as the set of requirements, no more, no less.

Since a formal specification is based on a logical system , completeness of specifications can be defined formally in the context of logic [AK94]:
Let T be a theory characterized by types $t_1, .., t_n$. Then, T is complete if and only if it can determine whether or not $T \vdash A$, for any formula A in the language that is characterized by types $t_{a1}, .., t_{an}$, where $\{t_{a1}, ..t_{an}\} \subseteq \{t_1, ..t_n\}$

Another application of formal specification is to document already existing implementation. Specification has to convey all the observable behaviors of the software module it specifies.

Consider the definition of a complete specification as it is addressed in Cohen et al [CH87]:
Let Φ denote a formal system, defined as a pair (L, Cn), where L is the language generated by some syntax $\mathbf{syn}(L)$*, and Cn is the consequence closure operator for the system. The operator Cn enjoys the following properties:*

1. $\forall x \subset L \bullet x \subseteq Cn(x)$, *(each theory contained in its own closure);*
2. $\forall x, y \subseteq L \bullet x \subseteq y \rightarrow Cn(x) \subseteq Cn(y)$, *(monotonicity);*
3. $\forall x \subseteq L \bullet Cn(x) = Cn(Cn(x))$, *(closure is maximal);*

A specification S in formal system $\Phi = (L, Cn)$ is a set of formulae in the language. Therefore, $S \subset L$. A specification S' in Φ' implements a specification S in Φ (written S' **impl** S), where $L \subseteq L'$, iff $Cn(S) \subset Cn(S')$. A complete specification is one that fully determines all its implementations. It means that the specification is so fully defined that no implementation which has more or less behavior (as observed from the formal system of specification) can be constructed. The same can be stated formally as follows:

A specification S is complete iff
$\forall S'$ **impl** $S \rightarrow Cn'(S'/\mathbf{syn}(S)) = Cn(S)$, where $(S'/\mathbf{syn}(S))$ means S' restricted to the syntax of S.
Informally, the above formula states that a consequence closure of any implementation must be evaluated to that of the specification.

The completeness definitions presented above are ideal cases. Even first order logic is only semi-decidable. For example, we can not decide, in general, if a consequence closure of one theory is a subset of a consequence closure of another theory. Therefore, determining if the theory is complete, in general, is an undecidable task. As a result, in practice only weaker definitions of completeness are employed. They address completeness of specification with respect to some suitable criterion. The choice of criterion depends on the particular task for which formal specification is used. Often a weaker completeness is also referred to as **sufficient completeness**.

Sufficient completeness criterion, as it is stated by Kaizhi Yue [Kai87] targets matching of system requirements, stated informally, or intuitive understanding of system behavior with the formal specification. This kind of completeness is also called an **external completeness**. To show that a specification is sufficiently complete we have to select a set of goals and show that these are consequences of the theory specified. The goals are selected based on the specifier's intuition about important features from the application domain. For example, if we specify behavior of a file, one of the goals would be the ability to read from a file the same information as was written into it, if reading is performed over the location in the file where the last writing was done.

Guttag and Horning[GH78] provide their definition of sufficient completeness for algebraic specifications. An algebraic specification is used to specify an abstract data type. This data type is called *distinguished sort*. The functions that return the values of the specified distinguished sort are called *generators*; and the functions that return other values are called *behaviors*. Apparently, the generators are used to generate new values of the distinguished sort and the inspectors are used to inspect the properties of the distinguished sort values. Specification is sufficiently complete if all the properties of the distinguished sort can be derived from the specification. To achieve it, every

behavior function has to be defined over the entire function's domain. This kind of completeness is also called an *internal completeness.* A heuristic to ensure internal completeness is a specification to contain axioms where every behavior is applied to every generator.

2.2 The completeness criterion for a black-box reuse class specification

The recent master's thesis research by Piero Colagrosso [Col93] outlined the completeness criterion for a specification, intended for a black-box class reuse. This criterion will be the subject for further development in this thesis, resulting into a methodology for the completeness verification and the algorithm to carry out this methodology semi-automatically. We will also address the aspects of incompleteness in the specifications, providing means to identify the incompleteness and rectify it.

2.2.1 Definition of a class behavior

The completeness criterion presented here is based on the trace assertion specification technique [BP86]. Consider a class C having a finite set of member-functions, partitioned as: a set of class constructors $C = \{c_1, \ldots, c_n\}$ and a set F of other functions $F = \{f_1, f_2, \ldots, f_n\}$. The trace of function invocations for a class C is any sequence of function calls in which the first item is a class constructor and the other items are member-functions but not constructors. Formally, a trace $Tr = c_n; S$, where $c_n \in C$ and $S = f_n; S$, where $f_n \in F$ and ; is a concatenation sign.

For the purpose of software reuse the principal type of incompleteness which is important to avoid is *partial specification* of the class behavior [AK94]. The behavior of the class is characterized by how the class responds to the invocation of its member functions. In order to reuse a class, the reuser has to have this information. Reaction of a class to invocation of its member-function also depends on the state of the class at the time of the invocation. On the other hand, the state of a class can be modified only by the class member-functions. In addition, we have to consider

that the member-function invoked will be executed only in cases when the function's precondition holds. If all member-functions called during the course of an arbitrary trace Tr were executable then the trace Tr is *Legal*, otherwise the trace Tr is *Illegal*.

As follows from the above observation, a class specification has to be able to answer the following questions:

- Is a trace Tr *Legal* or *Illegal* for any arbitrary trace Tr?
- Let a member function f be invoked in the course of some legal trace Tr. What is the response of the class to this invocation?

2.2.2 Definition of the completeness criteria

To understand different responses of a class to invocation of its member-function, we categorize the effect of this response as follows:

- **V-action:** A method which returns a value belongs to this category.
- **S-action:** A method which produces an observable side effect in the object's external environment (e.g. display a window, modify color of a screen, transmit packet on a network) belongs to this category.
- **O-action:** A method which changes the abstract state of an object belongs to this category.

An abstract state of an object can be observed only by values returned from the calls to the member-functions of the class. Thus, result of an O-action can be perceived only by calls to the member-functions which belong to V-action or S-action. As a result, we consider only V-action and S-action to be the ones concerned with a class interface behavior. With the above categorization of the effect of the member-function invocation we present the completeness criteria as follows:
A class specification is complete iff the following conditions are satisfied :

1. For any given trace of a class, it is possible to determine whether or not the trace is legal.

2. For every legal trace ending with a call to a V-method, the abstract value returned can be derived from the specification.

3. For every legal trace ending with a call to an S-method, the side effect which will occur as a result of this last method execution can be derived from the specification.

To make the completeness criteria more formal, we introduce the following definitions:

- *LegalTr(C)* is a set of all the legal traces which can be constructed from the member-functions of a class C.

- Av is a function which maps a legal trace Tr ending by a call to a V-method into the abstract value returned by this method.

- As is a function which maps a legal trace Tr ending by a call to an S-method into the state of the external environment, as the latter is observed from the class C.

Finally, we state the completeness criteria for a specification Sc of a class C intended for a black-box reuse as follows:
Specification Sc is complete if it can be used to derive the behavior triple $(LegalTr(C), As, Av)$ of the class C.

2.3 Larch/C++ - Quick overview

Larch/C++ [leava] is a part of the Larch family of specification languages [GH78]. Larch languages are formal specification languages geared towards the specification of the observable effects of program modules, particularly modules which implement abstract data types; Larch provides a two-tiered approach to specification:

- In one tier, a Larch Interface Language (LIL) is used to describe the semantics of a program module written in a particular programming language. LIL specifications provide the information needed to understand and use a module

interface. LIL doesn't refer to a single specification language but to a family of specification languages. Each specification language in the LIL family is designed for a specific programming language.

LIL specifications are used to specify the abstract state transformations resulting from the invocation of the operations of a module. These specifications are written in a predicative language using pre- and post-conditions.

- In the other tier, the Larch Shared Language (LSL) is used to specify state-independent, mathematical abstractions which can be referred to in LIL specifications. These underlying abstractions, called *traits*, are written in the style of an equational algebraic specification.

  LSL is a programming language independent from and shared by all LILs.

Larch/C++ is an interface specification language for specifying C++ classes and functions. The restriction to C++ allows Larch/C++ to have a syntax and semantics that is tailored to C++; for example, the Larch/C++ specification of a C++ function specifies not only the behavior of the function, but exactly how that function is called from C++ code. The details of how to call a C++ function, the name, return type, and argument types, are called the interface of the function.

Interface specifications rely on definitions from auxiliary specifications, written in LSL, to provide a semantics for the primitive terms they use. Larch encourages a separation of concerns, with basic constructs in the LSL tier and programming details in the interface tier.

Functions are specified in Larch/C++ using Hoare-style pre- and post- conditions. The header of a function specification is the same as that of C++ function definitions. The body describes the effect of function invocation using a pair of predicates following the keywords **requires** and **ensures**. The predicate following **requires** is a pre-condition that must be satisfied to invoke the specified function. The predicate following **ensures** is a post-condition that the specified function establishes upon termination. The notations '$\wedge$' and '$\vee$' denote conjunction and logical disjunction respectively. All the logic notations such as $\forall$ - universal quantifier, $\exists$ - existential

quantifier are also valid. The semantics of function specification is that the pre-condition of the state transformation must logically imply the post-condition of the state transformation. For functions that change the values of objects, the body of the function specification must include a **modifies** clause. Only objects listed in the **modifies** clause are allowed to change their values as the result of function invocation. In a function that mutates an object or a variable, there are two different values for the same object; the value in a pre-state and the one in the post-state. The value of the object in the pre-state is denoted by a hat-ed (^) identifier, while the post-state value is represented by a primed (') identifier. If neither of (^) nor (') is used with the object name then the object itself is considered as a memory location and not the object value.

The syntax for data members and member functions in interface specifications are almost the same as in a C++ program. The Larch/C++ reserved word *this* is used in the member function specifications and means the same thing as the C++ reserved word *this*, a pointer to the object of the specified class. The Larch/C++ reserved word *self* is a shorthand for $*(this\,\backslash any)$. The suffix *any* is like (') or (^), and extracts the value of *this* in some visible state. As in C++, Larch/C++ member functions can be public, protected, and private.

Figure 1 shows a Larch/C++ class interface specification for the class Set. The **uses** clause indicates that the Larch/C++ interface is expressed with the vocabulary of the LSL trait `SetTrait` (see Figure 2). The trait `SetTrait` defines the terms used to denote abstract values of the set as well as the mathematical properties of the set. All the terms in the pre- and post- conditions of the function specifications come from this trait. The type-to-sort mapping, which is given between the parentheses following the names of the used trait, says that the abstract values of the C++ `Set` objects are specified to be those of the LSL sort C in `SetTrait`. (In LSL a sort is the type of an LSL term; the word type is used only to refer to C++ types). The type to sort mapping makes the connection between the C++ world and the LSL (mathematical) world. The abstract values of Set objects are denoted by equivalence classes of LSL terms of sort C from `SetTrait`.

Figure 1 specifies a constructor, a destructor, and three public member functions: *insert*, *delete* and *isEmpty*. The destructor uses the Larch/C++ reserved word **trashed** to state that the object **self** is no longer available. The terms *insert*, *isEmpty* and *delete* which appear in the pre- and post-conditions refer to the LSL operators, not to the member functions having the same name. All C++ declarations are legal in Larch/C++ interface specifications; for example, member functions can be virtual, static, friend, or inline; they all have their C++ meaning. The Larch/C++ keyword **result** can only be used in post-conditions and it denotes the function return value. The sort of **result** is the sort associated with the return type specified for the function. For example, in the interface specification for class `Set` the member function **isEmpty** returns sort Bool.

```
class Set
{
    uses SetTrait(IntSet for C, int for E);
 public:
    IntSet()
    {
            modifies self;
            ensures self' = empty;
    }
    ~IntSet()
    {
            modifies self;
            ensures trashed(self);
    }
    IntSet& insert (int i)
    {
            modifies self,
            ensures self' = insert(self',i) ∧ result = self;
    }
    IntSet& delete (int i)
```

```
    {   .
            modifies self;
            ensures delete(self^, i) = self' ∧ result = self;
    }
    Bool isEmpty ()
    {
            ensures if isEmpty(self^) then
            result = TRUE else result = FALSE;
    }
};
```

Figure 1: Set

SetTrait(E, C) : **trait**

% Essential finite-set operators

introduces

$\{\} :\rightarrow C$

$insert : E, C \rightarrow C$

$delete : E, C \rightarrow C$

$\_ \in \_ : E, C \rightarrow Bool$

$isEmpty : C \rightarrow Bool$

asserts

C **generated by** $\{\}, insert$

C **partitioned by** $\in$

$\forall\ s : C, e, e_1, e_2 : E$

$\neg(e \in \{\})$

$e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$

$isEmpty(\{\})$

$\neg isEmpty(insert(e, s))$

$e \in s \Rightarrow \neg isEmpty(s)$

$delete(e, \{\}) == \{\}$

$delete(e_1, insert(e_2, s)) ==$ **if** $e_1 = e_2$ **then** s

else $insert(e_2, delete(e_1, s))$.

Figure 2: SetTrait

Chapter 3

A Methodology to Evaluate Completeness of Larch/C++ Specifications

This chapter presents a methodology to evaluate completeness of a C++ class interface specification written in Larch/C++. This methodology is based on the completeness criteria introduced in the previous chapter for C++ classes intended for black-box reuse. The methodology provides the steps to guide completeness verification. In later chapters these steps will be transformed into an algorithm and a semi-automated technique for completeness verification will be introduced.

3.1 Sufficient conditions for completeness of Larch/C++ specification

Larch/C++ specification Sc for a class C is complete with respect to the black-box reuse completeness criteria if the specification Sc can be used to construct the behavior triple (LegalTr(C), Av, As). In order to show when such a triple can be constructed we decompose the process into three steps, one for each component of the behavior triple. We will examine the conditions which are sufficient for the corresponding

element of the behavior triple to be derived from Sc.

3.1.1 Legality of the trace

The first component of the behavior triple (LegalTr(C)) states that the specification Sc can be used to construct all legal traces of class C. That is, for a given arbitrary trace t, we can determine if t $\in$ LegalTr(C) or t $\notin$ LegalTr(C) using Sc.

Let $Tr(n)$ denotes trace Tr of length n, $n \geq 1$. A trace $Tr(n)$ is legal if its sub-trace $Tr(n-1)$ is legal and the precondition of the n-th function invoked is $TRUE$. On the other hand, the trace $Tr(n)$ of length n is illegal if its sub-trace $Tr(n-1)$ is illegal or the precondition of the n-th function invoked is $FALSE$. Based on this definition we state the sufficient condition for specification to be able to determine whether or not a trace is legal.

i 1 *If the precondition of every function composing an arbitrary trace t can be evaluated to TRUE or FALSE based on specification Sc then Sc can be used to determine if t $\in$ LegalTr or t $\notin$ LegalTr; otherwise Sc fails to determine the legality of t and, hence, Sc is incomplete.*

It is not feasible to consider all the arbitrary traces of the class because their number is infinite. As a result, we will address sufficient conditions in **i 2** to establish **i 1**.

i 2 *If the precondition Pr of every function from Sc can be evaluated to TRUE or FALSE in arbitrary state of class*[1] *C and with arbitrary value of variables referred in Pr then Sc can be used to determine if t $\in$ LegalTr or t $\notin$ LegalTr; otherwise Sc fails to determine the legality of t and, hence, Sc is incomplete.*

Preconditions in Larch/C++ specifications are first order logic formulas. Such a formula is composed of one or more terms. First order logic formula can be evaluated if each of the terms composing the formula can be evaluated. Consequently, we restate **i 2** as **i 3**.

[1] Here after, the state of the class must be understood as the state of the object instantiated from the class.

i 3 *If every term in the precondition Pr of every function from Sc can be evaluated in any arbitrary state of class C and with any arbitrary value of variables referred in the term then Sc can be used to determine if* $t \in LegalTr$ *or* $t \notin LegalTr$*; otherwise Sc fails to determine the legality of t and, hence, Sc is incomplete.*

Any state of a class is identified by the abstract value of the class, encapsulated in the class variable *self*. Hence, the state of the class affects the evaluation of a term only if *self* variable is referenced in the term. With this in mind we restate **i 3** as **i 4**.

i 4 *If every term in the precondition Pr of every function from Sc can be evaluated with any arbitrary value of variables referred in the term then Sc can be used to determine if* $t \in LegalTr$ *or* $t \notin LegalTr$*; otherwise Sc fails to determine the legality of t and, hence, Sc is incomplete.*

Terms referenced in Larch/C++ specification are defined in the underlying LSL trait(s). Since we do not assume any particular time of function invocation nor any specific values of the parameters referenced in the function, t . satisfy **i 4**, terms referenced in a function's precondition have to be defined over the complete range of its parameters. In other words, each term must be a total function. Although an LSL trait can explicitly exempt definition of the term over some range of its parameters, stating this fact in the **exempting** clause, this exemption is not acceptable when evaluating **i 4** and must be treated as any other source of non-totality of the term.

Even if the term is a total function, the values of the term parameters must be known in order to evaluate the term. A term parameter is one of the following: a variable supplied as the function argument; a class variable *self*; global variables in the class. If the term parameter is a function argument then the value of the argument will be supplied when the function is invoked. On the other hand, if the term parameter is a global variable or state variable *self*, their values are not provided at the time of the function invocation, but rather encapsulated and maintained internally by the class.

Self and class global variables are also called internal variables. To guarantee a known value of the internal variable(s), we have to ensure that it can be derived

from the specification Sc. Instantiated by the class constructor, these variables are affected by the member functions during the execution of the trace t, if the variables are mentioned in **modifies** clause of the respected functions. The function modifies the internal variable as specified by the function's postcondition. Thus, in order to keep track of the values of the internal variables, every postcondition modifying them has to be evaluable. Since no assumption is made about the time of the function invocation nor the values of the parameters referenced in it, every term composing the postcondition of such a function must be a total function, except the range of the term variables when the function is never executed. This is the range when the precondition is $FALSE$ or when the term is referenced in the LSL **exempting** clause.

Consider a formula in the function postcondition modifying an internal variable of the class. It is possible that the formula does not provide a deterministic value of the internal variable even if all the terms composing the formula are total functions. For example, the postcondition of the member-function in a Set specification can be written like this:

$self' = insert(self, e) \vee self' = self$

Obviously we do not know the value of $self'$ after execution of such a function because the postcondition does not provide us with a deterministic answer. Since known values of the internal variables are required after execution of any arbitrary trace t, functions which modify the internal variables must modify them deterministically. In the light of the above observations we restate **i 4** as **i 5**

i 5 *Sc can be used to determine whether* $t \in LegalTr$ *or* $t \notin LegalTr$ *if:*

- *Every term in the precondition of every function from Sc is a total function;*
- *Every term in the postcondition of every function modifying internal variable is a total function except the range of the terms parameter variables when the function's precondition is* $FALSE$ *or the range when the term is exempted in corresponding LSL trait;*
- *Postcondition of every function modifying an internal variable produces deterministic values after such a modification;*

3.1.2 Function's return values

ii 1 *Specification must be able to identify values returned by a V-method invoked after execution of the arbitrary legal trace t.*

The value returned by a V-method in Larch/C++ is either:

- The value of the *result* variable referenced in the corresponding function

or

- The value of the function parameters, modified after function execution.

Because the postcondition of the function specifies the function returned values, we must be able to evaluate the postcondition of a V-method after execution of an arbitrary trace t. Since the postcondition is a first order logic formula, the postcondition can be evaluated if the terms in the postcondition can be evaluated individually.

It is not feasible to consider all the possible legal traces of the class. Instead we address the invocation of the V-method at any time during the existence of a class. Such consideration is sufficient with respect to requirements stated in **ii 1**.

Since we consider invocations of the V-method at any time of the class's existence, parameter variables can have any arbitrary values. As a result of the above observations, we restate **ii 1** as:

ii 2 *Terms of any V-method occuring in the postconditions have to be specified in such a way that it is possible to evaluate them at any time during function execution.*

It is worth noticing that a function is executed only if its precondition is evaluated to $TRUE$ and none of the terms composing the postcondition fall into *exempting* LSL clause.

To enable evaluation of the term, values for the parameters referenced in the term must be known. Term parameter is either a function parameter or an internal variable: the global variables or the class variable $self$. As was shown in the previous section, the values of the internal variables must be derivable from the class specification in order to guarantee known values of variables referenced by the function. This is satisfied when the following are true:

- Every term in the postcondition of every function modifying internal variable is a total function except the range when the function's precondition is $FALSE$ or the range when the term is exempted in corresponding LSL trait.

- The postcondition of every function modifying internal variables produces deterministic values resulting from such a modification.

The above observations transform **ii 2** into:

ii 3 *For Sc to be able to identify values returned by a V-method it is sufficient if the following hold:*

- *Terms of any V-method, occurring in the postconditions are total functions except when the corresponding precondition is $FALSE$ and when one of the terms falls into corresponding exempting clause.*

- *Every term in the postcondition of every function modifying an internal variable is a total function except when the function's precondition is $FALSE$ or when the term is exempted in corresponding LSL trait.*

- *The postcondition of every function, modifying internal variables, produces deterministic values resulting from such a modification.*

Since all member functions of a class are totally partitioned by V-methods and methods that modify internal variables, we can summarize items of **ii 3** as follows:

ii 4

- *Every term in the postcondition of every function is a total function except when the function's precondition is $FALSE$ or when the term is exempted in the corresponding LSL trait.*

- *The postcondition of every function, modifying internal variables, produces deterministic values resulting from such a modification.*

3.1.3 Environment change

An S-method is a one that changes the state of the class environment. The last element of the behavior triple states that specification must provide the information to derive the effect of the S-method(s) executed in an arbitrary legal trace t. Some examples of S-method are: changing the position of the cursor in the screen, filling a structure encapsulated in the class and sending a packet to the network. Only changes in the environment that can be observed from our class are of our interest. In the context of Larch/C++ this addresses modification of the global variables in the class and the side effect of the function execution, resulting in the modification of the variables passed as arguments to the function.

By conducting an analysis analogous to the one from the previous section we concluded the following:

iii 1 *For Sc to be able to identify the state of the class environment which results after execution of the trait t containing S-method(s), it is sufficient if the following hold:*

- *Every term in the postcondition of every function modifying variables is a total function except when the function's precondition is FALSE or when the term is exempted in the corresponding LSL trait.*
- *The postcondition of every function modifying variables produces deterministic values resulting from such a modification.*

3.1.4 Summary

We have identified the requirements for each component of the behavior triple which Larch/C++ specification must satisfy in order for the component to be derivable from the specification. Now, we summarize these requirements to state the sufficient conditions of specification to be able to provide the behavior triple and, hence, to be complete.

Completness Requirements

Specification Sc is complete if :

1. Each term in the precondition of every function from Sc is a total function

2. Each term in the postcondition of every function is a total function except when the function's precondition is $FALSE$ or when the term is exempted in the corresponding LSL trait.

3. The postcondition of every function modifying variables produces deterministic values for the variables after such a modification

3.2 Proving Sufficiency

Here we present a proof that **Completeness Requirements** are sufficient conditions for a specification to be complete. We will prove that if **Completeness Requirements** are satisfied then each of the components of the behavior triple is satisfied and, hence, the whole triple is satisfied, implying the completeness of the specification.

Lemma 1 (Legality of the trace) *For every trace of a class, it is possible to determine whether or not the trace is legal.*

Proof by contradiction:

Assume **Completeness Requirements** are *TRUE* but **Lemma 1** is *FALSE*. If **Lemma 1** is *FALSE* then there is a trace t where the pre-condition of the last function in the trace cannot be evaluated.

Pre-condition is a first order logic formula. Evaluation of the first order logic formula is evaluation of the terms comprising the formula in some sequence which is identified by the form of the formula. In other words, one term is evaluated after another. If the formula cannot be evaluated, then there is a step of evaluation when the term can not be evaluated. The reason for the inability to evaluate the first order logic term is either there is a free variable in a term and the value of this variable is not defined or the term is not defined over the complete range of its parameters. The former contradicts both **Completeness Requirements(2)** and **Completeness Requirements(3)**, the latter contradicts **Completeness Requirements(1)**. Hence, Completeness Requirements are sufficient conditions for **Lemma 1**.

Lemma 2 (V-method) *For every legal trace ending with a call to a V-method, the abstract value returned can be derived from the specification.*

Proof by contradiction:

Assume **Completeness Requirements** are *TRUE* but **Lemma 2** is *FALSE.* If **Lemma 2** is *false* then there is a legal trace t, ending by a V-method, for which either the post-condition cannot be evaluated or there is an internal variable whose value is not known prior to the V-method invocation. This contradicts both **Completeness Requirements(2)** and **Completeness Requirements(3)** . Consider case when a post-condition cannot be evaluated. Post-condition is a first order logic formula. Evaluation of the first order logic formula is evaluation of the terms comprising the formula in some sequence which is identified by the form of the formula. In other words, one term is evaluated after another. Hence, if formula cannot be evaluated, then there is a step of evaluation when the term can not be evaluated. The reason for inability to evaluate the term is either there is a free variable in a term and the value of this variable is not defined or term is not defined over the complete range of its parameters. Since from initial assumption the trace t is legal, the range of the term where the term is not defined is different from both: the range satisfying expression $precondition = FALSE$ and the range identified by the term **exempting** clause. The former contradicts both **Completeness Requirements(2)** and **Completeness Requirements(3)**, and the latter contradicts **Completeness Requirements(2)**. Hence, **Completeness Requirements** is a sufficient condition for **Lemma 2.**

Lemma 3 (S-method) *For every legal trace ending with a call to an S-method, the side effect which will occur as a result of this last method can be derived from the specification.*

Proof by contradiction:

Assume **Completeness Requirements** are *TRUE* but **Lemma 3** is *FALSE.* If **Component 3** is *FALSE* then there is a legal trace t, ending by an S-method, whose side effect can not be derived from the specification. Since side effect of the function changes the function's parameter variables, our assumption implies that the values of the function's parameters can not be evaluated after execution of some function at the end of some legal trace t. The reason for the above fact could be one of the

following:

1. the value variables were not known prior to the function invocation
2. it is not possible to evaluate function's postcondition
3. function's post-condition specifies more then one possible value for the variable(s) it modifies

Consider (1). Since non-internal variables are supplied as function arguments and these arguments are assumed to be known, then it is an internal variable whose value was not known prior to the function invocation. This contradicts both **Completeness Requirements(2)** and **Completeness Requirements(3)** .

Consider (2). Since invocation of the regarded function is legal, function's precondition is $FALSE$ and neither of the terms in the function's post-condition falls into the **exempting** clause. This contradicts **Completeness Requirements(2)**.

Consider(3). It contradicts **Completeness Requirements(3)**. hence, we conclude that **Completeness Requirements** is a sufficient condition for **Lemma 3**.

Thus, we have proved that **Completeness Requirements** is a sufficient condition for each of the components of the behavior triple and therefore, they form a set of sufficient conditions for a specification to be complete.

Chapter 4

Realization of the methodology

In the previous chapter we identified the properties of a Larch/C++ interface specification which are sufficient conditions for the specification to be complete. In this chapter we will show how to develop an approach for the verification of these properties and provide algorithmic steps for it. The chapter is concluded by an example, where the methodology is applied to verify the completeness of a Larch/C++ specification.

4.1 Towards the Algorithm

Larch/C++ specification relies on first order logic theories to specify its terms and sorts. At the same time, Larch/C++ specification can not be expressed only by means of the first order logic theories because it uses **state machine** semantics. Fortunately, the properties of Larch/C++ specifications addressed in the completeness verification methodology are confined to the properties of the first order theories underlying the interface specification. This fact simplifies the completeness verification task and makes it possible to semi-automate it, using existing machinery in theorem provers. A number of proof assistants have been developed to work with first order logic theories. We will use Larch Prover (LP), the one that was built to work with Larch specifications. The following sections provide the steps necessary to prepare assertions and theorems to prove in order to realize the methodology presented in the previous

chapter.

4.1.1 Totality of terms in precondition

We have to establish that all the terms referenced in preconditions are total functions. In order to do so, we have to analyze the LSL trait, defining the term. The assertions defining the term must provide enough information for the term to be defined over the whole range of its parameter(s). As stated in the previous chapter the terms in a precondition should not have any exempted range. Therefore, we disregard statements in the **exempting** clause and address only statements from **assert** clause from the corresponding LSL specification.

A term is a mathematical function. Thus, it provides a unique mapping from its domain into range. Our goal is to ensure that there is no region in the term domain where the mapping is not specified.

Consider an arbitrary term $T : p1..pn \rightarrow r$, where T is the name of the term, $p1..pn$ is the list of parameters and r is the result produced. Let the term $T' : p1..pn \rightarrow r$ have the same behavior as term T. Behavior of the term is the way it maps the values from the term domain into the values from the term range. Thus, saying that the two terms have the same behavior means that they have the same domain, the same range, and the same mapping from the domain values into the range values. Assume that the behavior of T is defined over the complete range of its parameters p1..pn. Then T and T' must produce the same value r when the parameters' values are equal (refer Fig. 3):

i 1 $\forall p1..pn(T(p1..pn) = T'(p1..pn))$

Let the behavior of term T be not defined over some domain $Undef$. In this case only a partial equality between T and T' can be established, since nothing is known of how the T maps its domain values into the range values when the first falls into the $Undef$ (refer Fig.4):

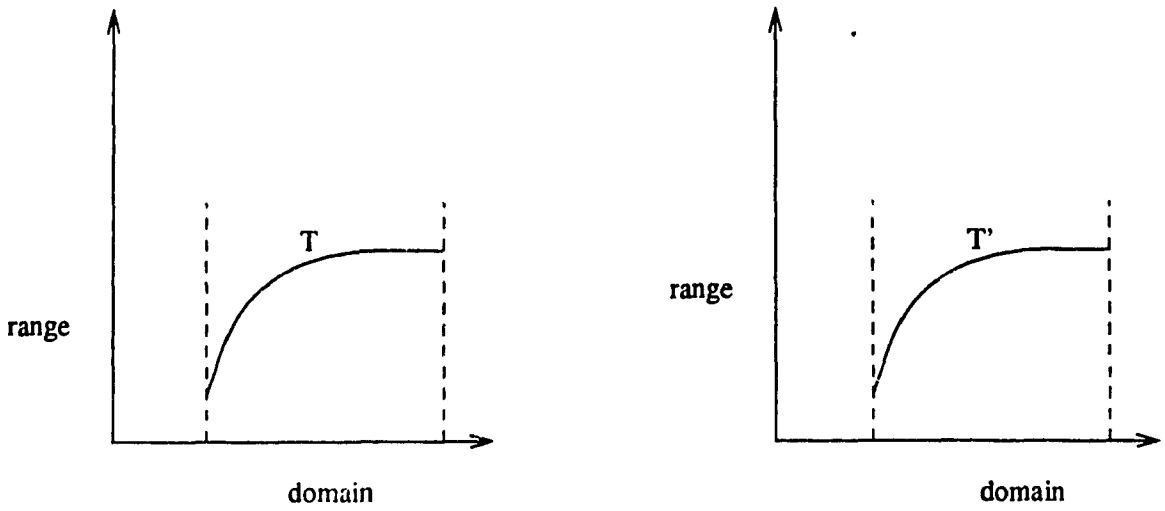

Figure 3: Total function

i **2** $\forall p1..pn((p1 \notin Undef \vee p2 \notin Undef ... \vee pn \notin Undef) => T(p1..pn) = T'(p1..pn))$

As a result, it is impossible to establish i **1**.

The above technique can be used to prove that the term is a total function. Let T be defined in the Larch trait Tr.

The first step is to create a term T' as having the behavior identical to T. In order to do so, we add a new signature to Tr which is the same as for T except that the name of T is substituted by the name of T' in the signature. We have to specify behavior of T' to be the same as that of T. In order to do so, for every assertion A referring to T we add a new assertion A' into Tr which is an exact copy of A except that every occurrence of T is substituted by T'. Now Tr contains specification of two terms T and T' with identical behavior. In order to show that T is a total function we have to prove $\forall p1..pn(T(p1..pn) = T'(p1..pn))$.

We have to notice that proving the totality of the function makes sense only if the function is not a basic constructor of a sort . Since by definition any sort value can be expressed in terms of basic constructors and the formula containing only basic constructors is not further reducible, a basic constructor is always a total function and any attempt to argue about it leads to the circular reasoning.

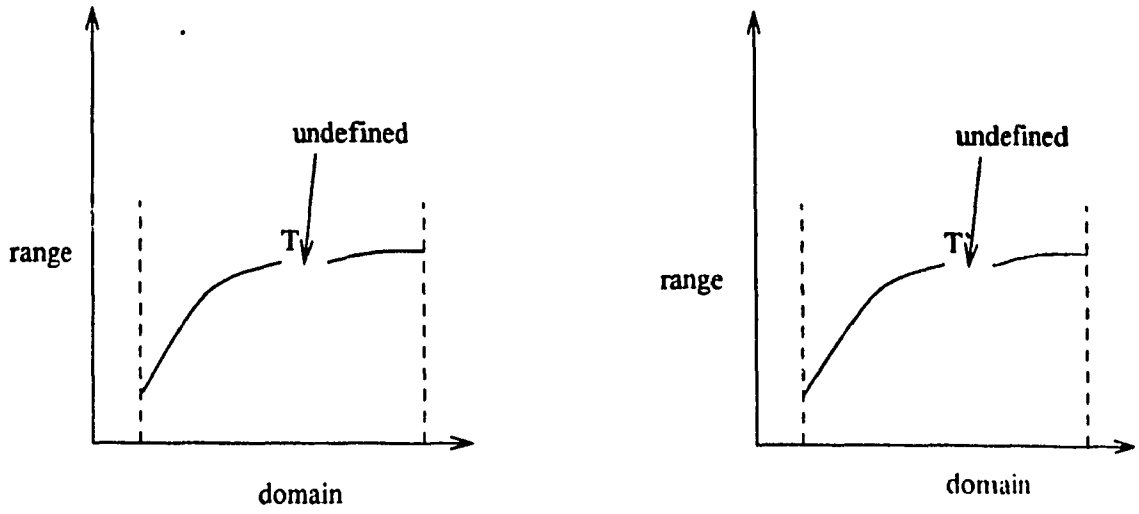

Figure 4: Partial function

4.1.2 Totality of the terms in postcondition

We have to establish that all the terms referenced in postconditions are total functions, except when the term falls into LSL *exempting* clause or when the function's precondition is $FALSE$. In the previous section we have already shown the technique to verify totality of the term. With slight modifications this technique is applicable to our current goal as well. When proving the totality of the term we have to incorporate the fact that there is a range where we do not want to verify whether or not the term is defined.

Consider an arbitrary term $T : p1..pn \rightarrow r$. To establish that T is a total function, we proceed as we did in the previous section: we define $T' : p1..pn \rightarrow r$ having the same signature as T and incorporate all the assertions referring to T, but stating them about T'. As before we have to prove the theorem:
$\forall p1..pn(T(p1..pn) = T'(p1..pn))$

In addition, we conjoin a new assertion:
$\neg Pr => (T(p1..pn) = T'(p1..pn))$, where Pr is the function's precondition.
This assertion states that over the range when the precondition of the function is $FALSE$ we do not need to verify whether the term's behavior is defined or not. Our proof obligation is exempted by this explicit assertion over the identified range.

We also add the assertions for each appearance of the term in the *exempting* clause:

$T(ExemptedRange) == T'(ExemptedRange)$

As in the case with **precondition**, we explicitly state that T and T' are equal over the *ExemptedRange* and our proof obligation is exempted over this range.

4.1.3 Determinism of variable modification

For every variable modified after execution of the function we have to ensure that it is modified deterministically.

First we have to determine which variables are the subject for modification. This information is explicitly stated in the **modifies** clause of the respective function. To ensure determinism of the variable modification the following must hold:

Determinism Theorem 1

- Let the function parameter Par be modified as a result of the function execution.
- Let the value of the variable $Vr1$ be equal to the value of the variable Vr prior to the function execution.
- Let Vr and $Vr1$ be passed as parameters Par at different times.
- The postcondition of the function ensures Vr and $Vr1$ to be assigned the same values provided that the values of other parameters remain the same.

Thus, if the function's post-condition is a formula:

$Post(Par1^{\wedge}, Par2^{\wedge}, ..Parn^{\wedge}, Par1', Par2', ..Parn')$

where $^{\wedge}$ identifies value of the parameter variable in the pre-state and $'$ identifies value of the parameter variable in the post-state, then in order to ensure deterministic effect of this post-condition on Par1 we have to prove that:

Determinism Theorem 2 $(Post(Par1^{\wedge}, Par2^{\wedge}, ..Parn^{\wedge}, Par1', Par2', ..Parn') \wedge$
$Post(Par1^{\wedge}, Par2^{\wedge}, ..Parn^{\wedge}, Par11', Par2', ..Parn')) => Par1' = Par11'$

For example, consider a post-condition:

$self' = first(self^{\wedge})$

To ensure that $self$ is modified deterministically, we have to prove the theorem:
$(self' = first(self\hat{}) \wedge self1' = first(self\hat{})) => self' = self1'$

4.2 The Completeness Verification Algorithm

In the previous sections we have already shown how to approach completeness verification methodology. Now we will summarize it in the completeness verification algorithm, which shows all the steps necessary to build assertions and theorems in order to apply completeness verification methodology.

Completeness verification algorithm

Given : Larch/C++ specification Sc.

Goal : prepare LSL traits with theorems in order to establish completeness of Sc.

Steps :

- for every member-function Fun from Sc.

  1. create LSL trait *CompScFun.lsl*.
  2. – include into *CompScFun.lsl* the traits identified in the *uses* clause of Sc.
     – include into *CompScFun.lsl* the traits instantiated as a result of mapping of built-in and defined C++ types, used in Sc, into LSL sorts.
  3. For every term T composing the **requires** clause which is not mentioned in the LSL **generated by** clause.
     (a) Add term T' to the CompScFun.lsl such that the signature of T' is identical to the signature of T except the term name, which is changed to the name of T'.
     (b) Find all assertions referring to T in the *CompScFun.lsl* included traits(this applies to statements in **partitioned by** clause as well) and for each assertion A.

i. Form assertion A' to be identical to A, except that for every occurrence of T the name of term T is substituted by one of T'.

ii. Add assertion A' to CompScFun.lsl.

4. Add the theorem $T' = T$ to CompScFun.lsl.

5. For every term T composing the **ensures** clause, which is not mentioned in the LSL **generated by** clause and does not occur the **requires** clause.

   (a) complete 3(a), 3(b) and 4.

   (b) Find all exemptions from **exempting** clause
   referring to T in the *CompScFun.lsl* included traits and for each exemption E.

      i. Form assertion A' as E=E', where E' is identical to E, except that in every occurrence of T in E' name of term T is substituted by one of T'.

      ii. Add assertion A' to CompScFun.lsl.

   (c) Add assertion $\neg Pre => T = T'$, where Pre is an expression in the **requires** clause of the function.

6. For each variable Vn mentioned in the **modifies** clause.

   (a) Form the theorem as
   $(Post(V1\hat{},..Vn\hat{},..Vm\hat{},V1',..\underline{Vn'},..Vm')\wedge$
   $Post(V1\hat{},..Vn\hat{},..Vm\hat{},V1',..\underline{Vn1'},..Vm')) => Vn' = Vn1'$, where
   $Post$ is an expression in the **ensures** clause of the respective function;
   $V1..Vm$ are variables referred by $Post$;
   $\hat{}$ and ' denotes values of the variable before and after execution of the function respectively.

4.3 An example

To illustrate the completeness verification algorithm, developed in this chapter we consider Larch/C++ specification for **class IntStack**. The interface specification is shown in Fig.5 and the LSL specification for sort Stack is shown in Fig.6.

First, consider LSL traits that we have to include into each *ComplStackFun.lsl*. According to the algorithm we have to include traits identified in the **uses** clause of our interface specification. There is only one such trait:
StackTrait(*IntStackforStack*, *intforE*).
Second, we have to include traits mapping used in the interface specification built-in and defined C++ types. Since there is no such type in the example, our **include** clause consists only of information extracted from the interface specification **uses** clause.

First function in Stack.lcc is the class constructor *IntStack*. Following **Step 1** and **Step 2** of the algorithm we create trait ComplStackIStack.lsl and form its **includes** clause as described in the previous paragraph. There is no **requires** clause in this function, so we proceed with **Step 5**. **Ensures** clause consists of only one term *new*, which is a basic constructor of Stack sort and therefore, it does not require any processing. Consequently, we proceed with **Step 6** of the algorithm and state the theorem in the **implies** clause of ComplStackIStack.lsl. The trait is shown in Figure 7.

Second function is the Stack.lcc is a class destructor. Since it uses only built-in LSL term *trashed*, we will not process this function, assuming that the built-in term has a complete specification.

For the remaining four functions of the class *Ipush*, *IisEmpty*, *Itop* and *Ipop* we built the traits according to the Completeness verification algorithm and show them in Figure 8, 9, 11 and 10 respectively.

```
class IntStack
{
    uses StackTrait(IntStack for Stack, int for E)
public:
    IntStack()
    {
            modifies self;
            ensures self' = new;
    }
    ~ IntStack
    {
            modifies self;
            ensures trashed(self);
    }
    void Ipush(int i)
    {
            modifies self;
            ensures self' = push(self^, i);
    }
    bool IisEmpty()
    {
            ensures result = isEmpty(self^);
    }
    int Itop()
    {
            requires not(isEmpty(self^));
            ensures result = top(self^);
    }
    void Ipop()
    {
            requires not(isEmpty(self^));
            modifies self;
            ensures self' = pop(self^);
    }
};
```

Figure 5: Integer Stack interface specification Stack.lcc

StackTrait$(E, Stack)$: **trait**
  % Essential LIFO operators
  includes *Integer*
  introduces
    $new :\rightarrow Stack$
    $push : Stack, E \rightarrow Stack$
    $top : Stack \rightarrow E$
    $pop : Stack \rightarrow Stack$
    $isEmpty : Stack \rightarrow Bool$
  asserts
    Stack **generated by** $new, push$
    Stack **partitioned by** $top, pop, isEmpty$
    $\forall\ e : E, s : Stack$
      $top(push(s, e)) == e$
      $pop(push(s, e)) == s$
      $isEmpty(new)$
      $\neg isEmpty(push(s, e))$

Figure 6: Stack.lsl

CompIStackIStack : **trait**
  includes *StackTrait*(*IntStack* **for** *Stack*, *Int* **for** E)
  implies
    $\forall$
      $self', self1' : IntStack$
      % determinism
      $(self' = new \wedge self1' = new) \Rightarrow (self' = self1')$

Figure 7: Constructor completeness verification

CompIStackIpush : **trait**
  includes *StackTrait*(*IntStack* **for** *Stack*, *Int* **for** E)
  implies
    $\forall$
      $self', self1', self : IntStack, i : Int$
      % determinism
      $(self' = push(self, i) \wedge self1' = push(self, i)) \Rightarrow self' = self1'$

Figure 8: Ipush completeness verification

ComplStackIisEmpty : **trait**
includes *StackTrait*(*IntStack* **for** *Stack*, *Int* **for** *E*)
introduces
$isEmpty' : IntStack \rightarrow Bool$
asserts
$\forall\ e : Int, s : IntStack$
$isEmpty'(new)$
$\neg isEmpty'(push(s, e))$
implies
$\forall\ e : Int, s : IntStack$
$isEmpty'(s) = isEmpty(s)$

Figure 9: IisEmpty member-function completness verification trait

ComplStackIpop : **trait**
includes *StackTrait*(*IntStack* **for** *Stack*, *Int* **for** *E*)
introduces
$isEmpty' : IntStack \rightarrow Bool$
$pop' : IntStack \rightarrow IntStack$
asserts
$\forall\ e : Int, s : IntStack$
$isEmpty'(new)$
$\neg isEmpty'(push(s, e))$
$pop'(push(s, \epsilon)) == s$
$isEmpty(s) \Rightarrow (pop'(s) = pop(s))$
implies
$\forall\ e : Int, s, self', self1', self : IntStack$
$isEmpty'(s) = isEmpty(s)$
$pop'(s) = pop(s)$
% determinism
$(self' = pop(self) \wedge self1' = pop(self)) \Rightarrow (self' = self1')$

Figure 10: Ipop member-function completness verification trait

$ComplStackItop$: **trait**
includes $StackTrait(IntStack$ **for** $Stack, Int$ **for** $E)$
introduces
$isEmpty' : IntStack \rightarrow Bool$
$top' : IntStack \rightarrow Int$
asserts
$\forall\ e : Int, s : IntStack$
$isEmpty'(new)$
$\neg isEmpty'(push(s, e))$
$top'(push(s, e)) == e$
$isEmpty(s) \Rightarrow (top'(s) = top(s))$
implies
$\forall\ e : Int, s : IntStack$
$isEmpty'(s) = isEmpty(s)$
$top'(s) = top(s)$

Figure 11: Itop member-function completness verification trait

Chapter 5

Description of proofs and completeness verification methodology

5.1 Overview of Larch Theorem prover

We already stated in the previous chapters that completeness verification algorithm can be carried out semi-automatically with the help of the machinery in the theorem provers. Since this thesis addresses Larch/C++ specifications, we will use the theorem prover designed to work with Larch Shared Language specifications.

Larch Prover(LP) [GG93] is a theorem prover for a subset of multi-sorted first-order logic. LP is intended as an interactive proof assistant rather than an automatic theorem prover. LP is designed with the assumption that the initial attempts to state the theorem correctly, and then to prove them usually fail. As a result, LP provides useful information about why proofs fail, if and when they do. This feature of LP is especially important when LP is used for the completeness verification. We need not only to be able to prove if the specification is complete, but, also, to have enough information to determine incompleteness, localize it and rectify. We will address this issue in the next chapter.

5.1.1. LP theories

A logical system is the basis for proofs in LP. This system consists of a set of declared operators, the properties of which are axiomatized by equations, rewrite rules, operator theories, induction rules, and deduction rules. The logical system in LP is a subset of multi-sorted first-order logic. Each kind of LP axiom has two semantics, a definitional semantics in first-order logic and an operational semantics that is sound with respect to the definitional semantics but not necessarily complete.

LP sort, operator and variable declarations are semantically the same as ones in LSL. LP has built-in sort *Bool*, as well as operators *true*, *false*, *if*, *not*, =, &(*and*), |(*or*), => (*implies*) , and <=> (*if and only if*). During a proof, LP can generate local variables, constants, and operators.

A term in multi-sorted first-order logic consists of either a variable or of an operator and sequence of terms known as its arguments. The number and sorts of the arguments in a term must agree with the declaration for the operator.

Equations

LP theory has equations in it. An equational theory is a theory (i.e., a set of facts) axiomatized by a set of equations. The syntax of equational theory can be defined in the following way. The set of terms constructed from a set of variables and operators is called *a free word algebra* or *term algebra*. A set S of equations defines a congruence relation on a term algebra. This relation is a smallest one that contains the equations in S and that is closed under reflexivity, symmetry, transitivity, instantiation of free variables, and substitution of equals for equals. An equation $t1 == t2$ is in the equational theory of S , or is an equational consequence of S, if $t1$ is congruent to $t2$.

Rewrite rules

To enable some of the LP inference mechanisms, equations have to be oriented into rewrite rules. Logical meaning of the rewrite rules is identical to that of equations. However, the operational behavior is different. A rewrite rule is an ordered pair (l, r) of terms, usually written as $l \rightarrow r$, such that l is not a variable and every variable

that occurs in r also occurs in l. A **rewriting system** is a set of rewrite rules. LP orients equations into rewrite rules and uses these rules to reduce terms to normal forms.

The reduction of terms to normal form can be described as follows. Define a *substitution* q to be a mapping from variables to terms such that $q(v)$ is identical to v for all but a finite number of variables. The domain of a substitution is extended to terms : $q(f(t1,...,tn))$ is defined to be $f(q(t1),...,q(tn))$. A substitution q matches a term $t1$ to a term $t2$ if $q(t1)$ is identical to $t2$. Each rewriting system R defines a binary relation $\rightsquigarrow_R$ (rewrites or reduces directly to) on the set of all terms. Operationally, $t \rightsquigarrow_R u$ if there is some rewrite rule $l \rightarrow r$ in R and some substitution q that matches l to a subterm of t such that u is the result of replacing that subterm by $q(r)$. The relation $\rightsquigarrow_R *$ is the reflexive transitive closure of $\rightsquigarrow_R$. Thus, $t \rightsquigarrow_R *u$ iff there are terms $t1,...,tn$ such that $t = t1 \rightsquigarrow_R ... \rightsquigarrow_R tn = u$. The relation $\rightsquigarrow_R +$ is the transitive irreflexive closure of $\rightsquigarrow_R$. It is usually essential that R is terminating. Thus, there is no infinite sequence $t1 \rightsquigarrow_R t2 \rightsquigarrow_R t3...$ of reductions.

Though in general it is undecidable whether the set of rewriting rules is terminating, LP provides a number of mechanisms that orient many sets of the equations into the terminating rewriting system. A term t is said to be *irreducible* if there is no term u such that $t \rightsquigarrow_R u$. If $t \rightsquigarrow_R *u$ and u is irreducible, then u is a terminal or normal form of t.

A term can have many different normal forms. Unless otherwise directed, LP keeps all rewrite rules and equations in normal form. If a rewrite rule or equation reduces to an identity, that is, to one in which the right and left hand sides have the same normal form, it is discarded.

If a term has only one normal form, it is called the **canonical form** of the term. A terminating rewriting system in which all terms have a canonical form is said to be **convergent**.

If a rewriting system is convergent, its rewriting theory (that is, the equations that can be proved by reducing them to identities) is identical to its equational theory. Most rewriting systems are not convergent. In these systems, the rewriting theory is a proper subset of the equational theory.

Operator theories

Some equations can not be oriented into terminating rewrite rules. These are associativity and commutativity statements. For example, attempt to orient commutativity $a + b == b + a$ into rewrite rules will produce non-terminating system: $a + b \rightarrow b + a; b + a \rightarrow a + b$. To avoid it LP uses **equational term-rewriting** to match and unify terms modulo associativity and commutativity. In equational term-rewriting, a substitution q matches $t1$ and $t2$ modulo a set S of equations if $q(t1) = t2$ is in the equational theory of S. For example, if + is ac (LP command to state associativity and commutativity), the rewrite rule $a * b \rightarrow c$ will reduce the term $a * c * b$ to $c * c$.

Inductive rules

Inductive rules increase the number of theories that can be axiomized using finite set of assertions. They have similar syntaxes and identical semantics to the inductive statements in LSL. An example is *Set* **generated by** *new*, *insert*. An equation in the set theory: $delete(insert(s, e), e) == s$ produces an infinite number of equations: $delete(insert(new, e), e) == new$); $delete(insert(insert(new, b), e), e) == insert(new, b)$ ... Thus, **generated by** clause is equivalent to the infinite set of first-order formulas: $(E[new] \wedge (\forall s : Set, b : element)(E[s] => E(insert(s, b)))) => (\forall s : Set)E[s]$, for any well formed equation E.

Deduction rules

LP uses deduction rules to deduce new equations from existing equations and rewrite rules. LP produces deduction rules from the LSL **partitioned by clause** . For example, LSL statement *Stack* **partitioned by** *isEmpty*, *top*, *pop* is reflected in LP theory as **assert when** $top(s1) == top(s2)$,
$pop(s1) == pop(s2)$,
$isEmpty(s1) == isEmpty(s2)$
yield $s1 == s2$.

5.1.2 Proof methods

LP provides mechanisms for proving theorems using both forward and backward inference. Forward inferences produce consequences from a logical system. Backward inferences produce a set of subgoals from a goal whose proof will suffice to establish a conjecture.

Normalization

Whenever a new rewrite rule is added to its logical system, LP renormalizes all equations, rewrite rules, and deduction rules. If an equation or rewrite rule normalizes to an identity, it is discarded. This is the way LP uses normalization in forward inference. Analogously, if a new conjecture is to be proved, LP tries to normalize it to an identity. If this attempt was successful then the conjecture is proved by normalization. The latter action of LP is a backward inference applying normalization.

Critical-pair equations

A common problem arises when a set of equations is oriented into a rewriting system, which is not convergent and, hence, there is more than one way to normalize the logical system. As a result, reduction to normal form does not provide a decision procedure for the equational theory. As a consequence, LP can fail, for example, to reduce term v and term u to the same normal form, even if $v \leadsto_R u$. LP might exhibit non-monotonic behavior; in other words, it may reduce u and v to the same normal form , using the rewriting system R but not using the system $R \cup \{l \rightarrow r\}$. The **critical-pair** command provides a method of extending the rewriting theory to more nearly approximate its equational theory. Each critical-pair equation captures a way in which a pair of rewrite rules might be used to reduce a single term in two different ways. For example, critical-pair equation between $(x*y)*z \rightarrow x*(y*z)$ and $i(w)*w \rightarrow e$ produces $e*z == i(y)*(y*z)$, when the substitution $\{i(y)\ for\ x, y\ for\ w\}$ unifies i(w)*w with subterm of $(x*y)*z$.

Instantiation

Explicit instantiation of variables in equations, rewrite rules, and deduction rules might lead to establishing that the conjecture is an identity. For example, to establish identity of the theorem $x == x \cup x$ in a logical system that contains the deduction rule **when** $(\forall e) e \in x == e \in y$ **yield** $x == y$ and the rewrite rule $e \in (x \cup y) \rightarrow e \in x | e \in y$, we instantiate y by $x \cup x$ in the deduction rule.

Proofs by cases

Conjecture can be often simplified by dividing a proof into cases. When a conjecture reduces to an identity in all cases, it is a theorem. For example, the command **prove** $0 < f(c)$ by case $c = 0$, will make LP to consider three cases: $c = 0$, $c < 0$, and $c > 0$. If in all three cases the conjecture is true then it is a theorem.

Proofs by induction

Proofs by induction are based on the induction rules, which we addressed earlier. The command **prove e by induction on x using** IR directs LP to prove the equation e by induction on variable x using the induction rule named IR. LP generates subgoals for the basic and inductive steps in a proof by induction as follows. The basic subgoals involves the equations that result from substituting the basic generators of IR for x in e. (Basis generators are those with no variables of the sort of x.) Induction subgoals generate one or more hypotheses by substituting one or more new constants for x in e. Each induction subgoal involves proving an equation that results from substituting a non-basic generator of IR (applied to these constants) for x in e (e.g., $insert(e, xc)$). For example, consider an induction proof over the sort Nat: **prove** $i \leq j => i \leq (j + k)$ **by induction on** j

Conjecture $lemma.1$:Subgoals for proof by induction on 'j'

Basis subgoal:

$lemma.1.1 : (i < 0) => (i < (0 + k)) == true$

Induction constant : jc

Induction hypothesis: $lemmaInductHyp.1 : (i < ic) => (i < (ic + k)) == true$

Induction subgoal:
lemma.1.2 : $(i < s(jc)) => (i < (s(jc) + k))) == true$

Proofs by implications

The command **prove** $t1 => t2$ **by** => directs LP to prove the subgoal t'2 using the hypothesis $t'1 == true$ (generally t'1=t1 and t'2 =t2, but in some cases LP has to generate new constants instead of variables in the t1 and t2 to form t'1 and t'2 and preserve soundness of proof). For example: Given the axioms $a => b \rightarrow true$ and $b => c \rightarrow true$, the command **prove** $a => c$ **by** => uses the hypothesis $a \rightarrow true$ to normalize the axiom and to reduce it to identity.

5.2 Proof methods and completeness verification

In general, proving theorems is an art. Thus, there is no universal recipe for conducting it. All the proof methods implemented in LP can be employed in one or another case. If an unrestricted variety of first order formulas is addressed, it is undecidable to generalize which proof method to apply and when. On the other hand, if only a certain kind of formula is considered, then one proof method is more likely to lead to success than another. Therefore, we can analyze and suggest the strategy to guide particular proof cases. Though this strategy helps to identify the proof methods, human assistance is still required to conduct a proof. As a result, full automation of a proof process cannot be achieved.

5.2.1 Strategy for proving totality of a term

Consider the two items from the Completeness verification methodology: proving totality of the terms from the **requires** and **ensures** clause. Since the goal is to ensure the totality of the function over the domain of its variables values, the most natural proof method to apply is proof by induction. Notice that the number of inductive steps does not exceed the number of variables in the term. Consider as an example an Integer Stack specification, presented in the chapter 3, Figure 3. Here we

show LP protocol of the proof obligations specified in ComplStackisEmpty.lsl trait, Figure 9 in Chapter 3:

```
LP0.1.15: prove
  isEmpty'(s) = isEmpty(s)
  ..


The current conjecture is ComplStackIisEmptyTheorem 1.

Conjecture ComplStackIisEmptyTheorem.1: isEmpty'(s) == isEmpty(s)
Proof suspended.

LP1: resume by induction on s

Conjecture ComplStackIisEmptyTheorem.1  Subgoals for proof by induction on 's'
Basis subgoal:
  ComplStackIisEmptyTheorem.1.1: isEmpty'(new) == isEmpty(new)
Induction constant  sc
Induction hypothesis:
  ComplStackIisEmptyTheoremInductHyp.1: isEmpty'(sc) == isEmpty(sc)
Induction subgoal
  ComplStackIisEmptyTheorem.1 2. isEmpty'(push(sc, e)) == isEmpty(push(sc, e))


The current conjecture is subgoal ComplStackIisEmptyTheorem 1 1.

Subgoal ComplStackIisEmptyTheorem 1.1: isEmpty'(new) == isEmpty(new)
[] Proved by normalization.

The current conjecture is subgoal ComplStackIisEmptyTheorem 1 2.

Added hypothesis ComplStackIisEmptyTheoremInductHyp 1 to the system

Subgoal ComplStackIisEmptyTheorem 1.2:
  isEmpty'(push(sc, e)) == isEmpty(push(sc, e))
[] Proved by normalization.

The current conjecture is ComplStackIisEmptyTheorem.1.

Conjecture ComplStackIisEmptyTheorem.1: isEmpty'(s) == isEmpty(s)
[] Proved by induction on 's'.
```

As another example, consider a proof for **RWFile** class specification, Figure 12 in Chapter 6. Here we present the proof of totality of *open* LSL function as a part of proof obligations for the class constructor, specified in CompFileConst.lsl trait,

Figure 14 in Chapter 6. The detailed analysis of **RWFile** class specification is in the next chapter.

```
LP0.1.16: prove
  open(f, m) = open'(f, m)
  ..


The current conjecture is CompFileConstTheorem.1.

Conjecture CompFileConstTheorem.1: open(f, m) == open'(f, m)
Proof suspended.

LP1. resume by induction on m

Conjecture CompFileConstTheorem.1: Subgoals for proof by induction on 'm'
Basis subgoals:
  CompFileConstTheorem.1.1: open(f, READ) == open'(f, READ)
  CompFileConstTheorem.1.2. open(f, WRITE) == open'(f, WRITE)
  CompFileConstTheorem 1.3: open(f, READ_WRITE) == open'(f, READ_WRITE)
The induction step is vacuous


The current conjecture is subgoal CompFileConstTheorem 1 1.

Subgoal CompFileConstTheorem.1 1. open(f, READ) == open'(f, READ)
[] Proved by normalization.

The current conjecture is subgoal CompFileConstTheorem.1.2.

Subgoal CompFileConstTheorem.1.2: open(f, WRITE) == open'(f, WRITE)
[] Proved by normalization.

The current conjecture is subgoal CompFileConstTheorem.1.3.

Subgoal CompFileConstTheorem.1.3: open(f, READ_WRITE) == open'(f, READ_WRITE)
[] Proved by normalization.

The current conjecture is CompFileConstTheorem.1.

Conjecture CompFileConstTheorem.1: open(f, m) == open'(f, m)
[] Proved by induction on 'm'.
```

Sometimes the inductive steps need additional user guidance. Completeness verification methodology exempts a term from the totality proof over the range of its variables values corresponding to the function precondition being $FALSE$. During

the course of proof by induction one or another deduction rule can be instantiated by the precondition expression so that LP can use instantiated formula to deduce totality of the term over the exempted range of the variables values. As a result, LP completes the inductive step over the entire range of the function domain.
To illustrate this point, consider *Itop* member-function from the **Integer Stack** example, Figure 5:

```
LP2: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  pop'(s) = pop(s)
  ..


The current conjecture is ComplStackIpopTheorem.2

Conjecture ComplStackIpopTheorem.2: pop'(s) == pop(s)
Proof suspended.

LP3: resume by induction on s

Conjecture ComplStackIpopTheorem.2: Subgoals for proof by induction on 's'
Basis subgoal.
  ComplStackIpopTheorem.2.1. pop'(new) == pop(new)
Induction constant: sc
Induction hypothesis:
  ComplStackIpopTheoremInductHyp 2  pop'(sc) == pop(sc)
Induction subgoal:
  ComplStackIpopTheorem.2.2. pop'(push(sc, e)) == pop(push(sc, e))


The current conjecture is subgoal ComplStackIpopTheorem.2.1.

Subgoal ComplStackIpopTheorem.2.1: pop'(new) == pop(new)
Proof suspended.

LP4: display ComplStackIpop

Rewrite rules:

ComplStackIpop.3: pop'(push(s, e)) -> s
ComplStackIpop.4: isEmpty(s) => (pop'(s) = pop(s)) -> true
```

```
/We will instantiate "s" by  "new" in ComplStackIpop.4
/to obtain the formula which is our current subgoal.
/We will use command  "make immune" to direct LP not to
/reduce obtained assertion to identity.

LP5: make immune ComplStackIpop.4

LP6. instantiate s by new in ComplStackIpop.4

Equation ComplStackIpop.4 has been instantiated to equation ComplStackIpop.4.1,
  pop'(new) = pop(new) == true

Deduction rule lp_equals_is_true has been applied to equation
ComplStackIpop 4.1 to yield equation ComplStackIpop 4 1.1,
  pop'(new) == pop(new), which implies ComplStackIpop 4.1.

Subgoal ComplStackIpopTheorem.2.1: pop'(new) == pop(new)
[] Proved by normalization

The current conjecture is subgoal ComplStackIpopTheorem.2.2.

Added hypothesis ComplStackIpopTheoremInductHyp.2 to the system.

Subgoal ComplStackIpopTheorem 2 2  pop'(push(sc, e)) == pop(push(sc, e))
[] Proved by normalization.

The current conjecture is ComplStackIpopTheorem 2.

Conjecture ComplStackIpopTheorem.2: pop'(s) == pop(s)
[] Proved by induction on 's'.
```

5.2.2 Strategy for proving the deterministic effect of variable modification

Third item in the completness proof methodology addresses the deterministic effect of the member-function on the modified variables. The equation built to prove this fact contains implication. Proof strategy in this case employs proof method **by implication**, possibly combined with some other methods. Usually, if the formula is not complex, success is achieved without additional interactions with the user. Consider, for example, the proof obligations for **RWFile** class trait CompFileRead1.lsl shown in Figure 22.

```
LP21: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (toByte(t') = __sel5_reddata(read(self, len(toByte(t)),
    __sel10_fpointer(self))) & __sel10_fpointer(self') =
    (__sel10_fpointer(self) + len(toByte(t))) & (toByte(t1') =
    __sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self))) &
    __sel10_fpointer(self') = (__sel10_fpointer(self) + len(toByte(t))))) =>
    t' = t1'
  ..


The current conjecture is CompFileRead1Theorem 4.

The equations cannot be ordered using the current ordering

Conjecture CompFileRead1Theorem.4
  ((__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
      = toByte(t'))
     & (__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
         = toByte(t1'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
    => (t' = t1')
    == true
  Current subgoal
    ((prefix(removePrefix(__sel4_data(self), __sel10_fpointer(self)),
        len(toByte(t)))
        = toByte(t'))
       & (prefix(removePrefix(__sel4_data(self), __sel10_fpointer(self)),
            len(toByte(t)))
            = toByte(t1'))
       & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
      => (t' = t1')
      == true
Proof suspended.

LP22:  resume by =>

Conjecture CompFileRead1Theorem.4: Subgoal for proof of =>
New constants: selfc, tc, t'c, t1'c, self'c
Hypothesis:
  CompFileRead1TheoremImpliesHyp.1:
     (prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
        len(toByte(tc)))
        = toByte(t'c))
       & (prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
```

```
            len(toByte(tc)))
            = toByte(t1'c))
        & ((__sel10_fpointer(selfc) + len(toByte(tc)))
            = __sel10_fpointer(self'c))
        == true
Subgoal:
  CompFileRead1Theorem.4.1: t'c = t1'c == true


The current conjecture is subgoal CompFileRead1Theorem.4.1.

Added hypothesis CompFileRead1TheoremImpliesHyp.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1 to yield the following equations, which imply
CompFileRead1TheoremImpliesHyp 1.
  CompFileRead1TheoremImpliesHyp 1.1.
     prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
       len(toByte(tc)))
       = toByte(t'c)
       == true
  CompFileRead1TheoremImpliesHyp 1.2.
     prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
       len(toByte(tc)))
       = toByte(t1'c)
       == true
  CompFileRead1TheoremImpliesHyp 1.3:
     (__sel10_fpointer(selfc) + len(toByte(tc))) = __sel10_fpointer(self'c)
       == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1.1 to yield equation
CompFileRead1TheoremImpliesHyp.1.1.1,
  prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
    len(toByte(tc)))
    == toByte(t'c),
which implies CompFileRead1TheoremImpliesHyp.1.1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2 to yield equation
CompFileRead1TheoremImpliesHyp.1.2.1,
  prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
    len(toByte(tc)))
    == toByte(t1'c),
which implies CompFileRead1TheoremImpliesHyp.1.2.

Deduction rule lp_equals_is_true has been applied to equation
```

```
CompFileRead1TheoremImpliesHyp.1.3 to yield equation
CompFileRead1TheoremImpliesHyp.1.3.1,
  __sel10_fpointer(selfc) + len(toByte(tc)) == __sel10_fpointer(self'c),
which implies CompFileRead1TheoremImpliesHyp.1.3.

Deduction rule Types.2 has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2.1 to yield equation
CompFileRead1TheoremImpliesHyp.1.2.1 1, t'c == t1'c,
which implies CompFileRead1TheoremImpliesHyp.1.2.1.

Deduction rule CompFileRead1 1 has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2 1 to yield equation
CompFileRead1TheoremImpliesHyp.1.2.1.2, t'c == t1'c,
which implies CompFileRead1TheoremImpliesHyp.1.2 1.

Subgoal CompFileRead1Theorem 4.1: t'c = t1'c == true
[] Proved by normalization

The current conjecture is CompFileRead1Theorem 4

Conjecture CompFileRead1Theorem.4:
  ((__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
      = toByte(t'))
     & (__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
         = toByte(t1'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
    => (t' = t1')
    == true
[] Proved =>.
```

If a formula has an implication on its top level, but the left hand side is rather a complex expression, it is worth trying to split the formula into a number of simpler cases and then apply proof by implication to each one of them. An additional reason for simplifying the formula befor applying the proof by implication is as follows. During the course of proof by implication, if the first attempt was not successful, we loose control over the part of the formula which was in the left hand side of the theorem. Consider a formula where prove by implication is applicable: $A \Rightarrow B$. LP assumes the left hand side A of the formula to be true: $A = TRUE$. Based on this hypothesis LP tries to derive $B = TRUE$, using the underlying axiom system. As a result, left hand side is not a goal any more but rather an assertion and we do not directly target it. Of course, it is possible to state subgoals manually and to try

to derive them from the hypothesis formed from the left hand side of our formula, but this is an additional loop in the course of the proof and it can be avoided. A natural way to obtain a number of simpler cases is to use proof **by cases**. It is desirable to identify trivial cases and conduct LP to consider them first. Consequently, these cases will be discharged without additional interactions with the user. At the same time, the cases which have not been discharged still have an implication in them and proof by implication can be applied to this simplified formulas. As an example consider the item from the proof obligations for bf RWFile class constructor shown in Figure 14:

```
LP34: prove
Please enter a conjecture to prove, terminated with a ' .' line, or '?' for
help.
  ((if(mode = 0.String, if(__sel7_name(f) = filename & opf = open(f,
    READ_WRITE), self' = opf, self' = open(create(filename, READ_WRITE),
    READ_WRITE)), self' = open(create(filename, mode), mode))) & (if(mode =
    0.String, if(__sel7_name(f) = filename & opf = open(f, READ_WRITE), self1'
    = opf, self1' = open(create(filename, READ_WRITE), READ_WRITE)), self1' =
    open(create(filename, mode), mode)))) => self' = self1'
  ..


The current conjecture is StringTheorem.8.

The equations cannot be ordered using the current ordering.

Conjecture StringTheorem.8.
  (if(0 = mode,
      if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
         opf = self',
         open(create(filename, READ_WRITE), READ_WRITE) = self'),
      open(create(filename, mode), mode) = self')
     & if(0 = mode,
          if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
             opf = self1',
             open(create(filename, READ_WRITE), READ_WRITE) = self1'),
          open(create(filename, mode), mode) = self1'))
    => (self' = self1')
    == true
  Current subgoal:
    (if(0 = mode,
        if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
             & (__sel7_name(f) = filename),
           opf = self',
           __mixfix2(__mixfix1(filename, empty, READ_WRITE),
             empty,
             READ_WRITE,
             1)
             = self'),
        open(__mixfix1(filename, empty, mode), mode) = self')
       & if(0 = mode,
            if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
                 & (__sel7_name(f) = filename),
               opf = self1',
               __mixfix2(__mixfix1(filename, empty, READ_WRITE),
```

```
                empty,
                READ_WRITE,
                1)
                = self1'),
            open(__mixfix1(filename, empty, mode), mode) = self1'))
      => (self' = self1')
      == true
Proof suspended.

LP35: resume by cases
Please enter terms defining cases, terminated with a '..' line, or '?' for
help:
(0:String=mode)=false
..


Conjecture StringTheorem.8: Subgoals for proof by cases
New constant: modec
Case hypotheses:
  StringTheoremCaseHyp.5.1: (0 = modec) = false == true
  StringTheoremCaseHyp.5.2: not((0 = modec) = false) == true
Subgoal for cases:
  StringTheorem.8.1:2:
     (if(0 = modec,
         if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
              & (__sel7_name(f) = filename),
            opf = self',
            __mixfix2(__mixfix1(filename, empty, READ_WRITE),
              empty,
              READ_WRITE,
              1)
              = self'),
         open(__mixfix1(filename, empty, modec), modec) = self')
        & if(0 = modec,
             if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
                  & (__sel7_name(f) = filename),
                opf = self1',
                __mixfix2(__mixfix1(filename, empty, READ_WRITE),
                  empty,
                  READ_WRITE,
                  1)
                  = self1'),
             open(__mixfix1(filename, empty, modec), modec) = self1'))
       => (self' = self1')
       == true


The current conjecture is subgoal StringTheorem.8.1.

Added hypothesis StringTheoremCaseHyp.5.1 to the system.

Deduction rule lp_not_is_true has been applied to equation
StringTheoremCaseHyp.5.1 to yield equation StringTheoremCaseHyp.5.1.1,
  0 = modec == false,
which implies StringTheoremCaseHyp.5.1.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.1:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
```

```
          & (__sel7_name(f) = filename),
        opf = self',
        __mixfix2(__mixfix1(filename, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
    & if(0 = modec,
        if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
            & (__sel7_name(f) = filename),
          opf = self1',
          __mixfix2(__mixfix1(filename, empty, READ_WRITE),
            empty,
            READ_WRITE,
            1)
            = self1'),
        open(__mixfix1(filename, empty, modec), modec) = self1'))
  => (self' = self1')
  == true
Current subgoal:
  ((open(__mixfix1(filename, empty, modec), modec) = self')
    & (open(__mixfix1(filename, empty, modec), modec) = self1'))
   => (self' = self1')
   == true
Proof suspended.

LP36  resume by =>

Subgoal StringTheorem.8.1: Subgoal for proof of =>
New constants: filenamec, self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp.2:
    (open(__mixfix1(filenamec, empty, modec), modec) = self'c)
      & (open(__mixfix1(filenamec, empty, modec), modec) = self1'c)
      == true
Subgoal:
  StringTheorem.8.1.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.8.1.1.

Added hypothesis StringTheoremImpliesHyp 2 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.2 to yield the following equations, which imply
StringTheoremImpliesHyp.2.
  StringTheoremImpliesHyp.2.1:
    open(__mixfix1(filenamec, empty, modec), modec) = self'c == true
  StringTheoremImpliesHyp.2.2:
    open(__mixfix1(filenamec, empty, modec), modec) = self1'c == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.2.1 to yield equation StringTheoremImpliesHyp.2.1.1,
  open(__mixfix1(filenamec, empty, modec), modec) == self'c,
which implies StringTheoremImpliesHyp.2.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.2.2 to yield equation StringTheoremImpliesHyp.2.2.1,
  open(__mixfix1(filenamec, empty, modec), modec) == self1'c,
which implies StringTheoremImpliesHyp.2.2.
```

```
Subgoal StringTheorem.8.1.1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.8.1.

Subgoal StringTheorem.8.1:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
         opf = self',
         __mixfix2(__mixfix1(filename, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
     & if(0 = modec,
          if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
               & (__sel7_name(f) = filename),
             opf = self1',
             __mixfix2(__mixfix1(filename, empty, READ_WRITE),
               empty,
               READ_WRITE,
               1)
               = self1'),
          open(__mixfix1(filename, empty, modec), modec) = self1'))
    => (self' = self1')
    == true
[] Proved =>.

The current conjecture is subgoal StringTheorem.8.2.

Added hypothesis StringTheoremCaseHyp.5.2 to the system.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.5.2 to yield equation StringTheoremCaseHyp.5.2.1,
  0 == modec, which implies StringTheoremCaseHyp 5.2.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.2:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
         opf = self',
         __mixfix2(__mixfix1(filename, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
     & if(0 = modec,
          if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
               & (__sel7_name(f) = filename),
             opf = self1',
             __mixfix2(__mixfix1(filename, empty, READ_WRITE),
               empty,
               READ_WRITE,
               1)
               = self1'),
```

```
          open(__mixfix1(filename, empty, modec), modec) = self1'))
    => (self' = self1')
    == true
  Current subgoal:
    (if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
          & (__sel7_name(f) = filename),
        opf = self',
        __mixfix2(__mixfix1(filename, empty, READ_WRITE), empty, READ_WRITE, 1)
          = self')
       & if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
             & (__sel7_name(f) = filename),
           opf = self1',
           __mixfix2(__mixfix1(filename, empty, READ_WRITE),
             empty,
             READ_WRITE,
             1)
             = self1'))
      => (self' = self1')
      == true
Proof suspended.

LP37: resume by cases
Please enter terms defining cases, terminated with a '..' line, or '?' for
help.
(__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
         & (__sel7_name(f) = filename)
..


Subgoal StringTheorem 8.2: Subgoals for proof by cases
New constants. fc, opfc, filenamec
Case hypotheses:
  StringTheoremCaseHyp.6 1:
     (__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
       & (__sel7_name(fc) = filenamec)
       == true
  StringTheoremCaseHyp.6.2:
     not((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
           & (__sel7_name(fc) = filenamec))
       == true
Subgoal for cases:
  StringTheorem 8 2.1.2:
     (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
           & (__sel7_name(fc) = filenamec),
         opfc = self',
         __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self')
        & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
              & (__sel7_name(fc) = filenamec),
            opfc = self1',
            __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
              empty,
              READ_WRITE,
              1)
              = self1'))
       => (self' = self1')
       == true
```

```
The current conjecture is subgoal StringTheorem.8.2.1.

Added hypothesis StringTheoremCaseHyp.6.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremCaseHyp.6.1 to yield the following equations, which imply
StringTheoremCaseHyp.6.1.
  StringTheoremCaseHyp.6.1.1:
     __mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc == true
  StringTheoremCaseHyp.6.1 2: __sel7_name(fc) = filenamec == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.6.1.1 to yield equation StringTheoremCaseHyp.6.1.1.1,
  __mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) == opfc,
which implies StringTheoremCaseHyp.6.1 1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.6.1.2 to yield equation StringTheoremCaseHyp 6.1.2.1,
  __sel7_name(fc) == filenamec,
which implies StringTheoremCaseHyp.6.1.2.

The equations cannot be ordered using the current ordering

Subgoal StringTheorem.8.2.1:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
       & (__sel7_name(fc) = filenamec),
     opfc = self',
     __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
   => (self' = self1')
   == true
  Current subgoal:
    ((opfc = self') & (opfc = self1')) => (self' = self1') == true
Proof suspended.

LP38: resume by =>

Subgoal StringTheorem.8.2.1: Subgoal for proof of =>
New constants: self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp.3: (opfc = self'c) & (opfc = self1'c) == true
Subgoal:
  StringTheorem.8.2.1.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.8.2.1.1.

Added hypothesis StringTheoremImpliesHyp.3 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.3 to yield the following equations, which imply
StringTheoremImpliesHyp.3.
```

```
  StringTheoremImpliesHyp.3.1: opfc = self'c == true
  StringTheoremImpliesHyp.3.2: opfc = self1'c == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.3.1 to yield equation StringTheoremImpliesHyp.3.1.1,
  opfc == self'c,
which implies StringTheoremImpliesHyp.3.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.3 2 to yield equation StringTheoremImpliesHyp.3.2.1,
  opfc == self1'c,
which implies StringTheoremImpliesHyp.3.2.

Subgoal StringTheorem 8 2.1.1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.8.2.1.

Subgoal StringTheorem.8 2.1:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
        & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
   => (self' = self1')
   == true
[] Proved =>

The current conjecture is subgoal StringTheorem.8.2.2.

Added hypothesis StringTheoremCaseHyp.6 2 to the system.

Deduction rule lp_not_is_true has been applied to equation
StringTheoremCaseHyp 6.2 to yield equation StringTheoremCaseHyp 6 2 1,
  (__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
    & (__sel7_name(fc) = filenamec)
    == false,
which implies StringTheoremCaseHyp.6.2.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.2.2:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
        & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
```

```
          1)
          = self1'))
    => (self' = self1')
    == true
  Current subgoal:
    ((__mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self')
      & (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self1'))
      => (self' = self1')
      == true
Proof suspended.
LP40: resume by =>

Subgoal StringTheorem.8.2.2: Subgoal for proof of =>
New constants: self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp 4:
     (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self'c)
       & (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
            empty,
            READ_WRITE,
            1)
            = self1'c)
       == true
Subgoal:
  StringTheorem.8.2.2.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem 8 2 2 1

Added hypothesis StringTheoremImpliesHyp 4 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.4 to yield the following equations, which imply
StringTheoremImpliesHyp 4.
  StringTheoremImpliesHyp.4.1:
     __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self'c
       == true
  StringTheoremImpliesHyp.4.2:
     __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self1'c
       == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.4.1 to yield equation StringTheoremImpliesHyp.4.1.1,
  __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
    == self'c,
which implies StringTheoremImpliesHyp.4.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.4.2 to yield equation StringTheoremImpliesHyp.4.2.1,
  __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
    == self1'c,
which implies StringTheoremImpliesHyp.4.2.
```

```
Subgoal StringTheorem.8.2.2.1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.8.2.2.

Subgoal StringTheorem.8.2.2:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
       & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
   => (self' = self1')
   == true
[] Proved =>.

The current conjecture is subgoal StringTheorem 8.2.

Subgoal StringTheorem.8.2.
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
        opf = self',
        __mixfix2(__mixfix1(filename, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
    & if(0 = modec,
         if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
              & (__sel7_name(f) = filename),
           opf = self1',
           __mixfix2(__mixfix1(filename, empty, READ_WRITE),
             empty,
             READ_WRITE,
             1)
             = self1'),
         open(__mixfix1(filename, empty, modec), modec) = self1'))
   => (self' = self1')
   == true
[] Proved by cases
   (__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
    & (__sel7_name(f) = filename).

The current conjecture is StringTheorem.8.

Conjecture StringTheorem.8:
  (if(0 = mode,
      if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
        opf = self',
        open(create(filename, READ_WRITE), READ_WRITE) = self'),
      open(create(filename, mode), mode) = self')
    & if(0 = mode,
```

```
        if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
           opf = self1',
           open(create(filename, READ_WRITE), READ_WRITE) = self1'),
        open(create(filename, mode), mode) = self1'))
   => (self' = self1')
   == true
[] Proved by cases (0 = mode) = false.
```

Chapter 6

Applying the completeness verification methodology

In this chapter we address Larch/C++ specification for a class from **Rogue Wave Tools.h++** [rogue] class library.

Rogue Wave Tools.h++ is a commercial product, used in Object Oriented software development. Classes implemented in the library have sufficient generality and a large set of methods which enables their software reuse. On the other hand, most of the library classes do not employ inter-object communication, providing a high level of functional and data encapsulation. The library is equipped with informal, but detailed documentation. These qualities promoted **Rogue Wave Tools.h++** as a bench mark for this thesis research.

In this chapter we present the analysis of the completeness verification of a class-sample from the library. Consequently, we address aspect of incompleteness of specifications, incompleteness localization and rectification. We will generalize the types of incompleteness and identify in which cases the incompleteness can not be removed.

6.1 Completeness verification of the RWFile class specification

Class RWFile[rogue] encapsulates file operations, using standard C stream library. The class interface offers general methods for a file manipulation. The interface specification of RWFile is shown in Figure 12, LSL specification of sort File is shown

in Figure 13.

The constructor has two arguments: *filename* and *mode*.
If *mode* is 0 then the constructor tries to find existing file with the name *filename* and open it for *READ_WRITE*. If this attempt fails , the constructor creates a file named *filename* and opens it for *READ_WRITE*. If *mode* is different from 0 then the constructor creates file *filename* and opens it with permission *mode*. The LSL trait CompFileConstractor.lsl contains constructor completeness proof obligations and is shown in Figure 14. All the conjectures stated in CompFileConstractor.lsl were proved successfully and, thereby, no incompleteness was detected at this stage. The protocol of the LP proofs is in Appendix, Figure 27.

The class destructor flushes contents of the file in memory into its image on disk, making the two identical. After that, the destructor closes the file. The LSL trait CompFileDestractor.lsl for the destructor completeness proof obligations is shown in Figure 21. The first attempt to prove the conjectures for this method was not successful. We failed to prove totality of the *flush()* LSL term. As a result, we analyzed the specification of the term *flush* in the File.lsl trait, Figure 13 and found that we did not state the effect of the function on the *OpenFile*. We rectified the incompleteness, changing the assertion as shown in File.lsl trait, Figure 13.

The proof obligation trait for method **Error** is shown in Figure 16. We failed to prove the conjectures stated in the trait. Apparently, the term *error()* lacks the assertions to enable the proof of the theorems stated. Moreover, there is no way to rectify the incompleteness detected. Indeed, the *error()* function is intended to express the fact that some error has occured during the lifetime of the OpenFile (e.g., input-output operation error). Obviously, occurrence of such an error is due to the circumstances which are external to the specified system . Therefore, an attempt to incorporate possible sources of the *error*() into the specification can not be successful. The assertions corresponding to *error*() are facts stating that some exceptions happened and, therefore, must be addressed in the exception handling part of the interface specification. These aspects will be considered in more detail in the next section.

No theorems for completeness proof obligation of the member-function **CurOffset, Erase, GetName, isEmpty, SeekTo, SeekToBegin** are needed to be proved. The proof obligation traits for member-functions **Eof, Exists, Flush, isValid** are shown in Figure 17, 18, 19, 20 respectively. The theorems from these traits are identical to those documented in the Appendix.

Proof obligations for method Read(&char c) are shown in Figure 22. The corresponding proof protocol is shown in Appendix, Figure 29.

Proof obligations for method Read(char* i, size_t count) are shown in Figure 23. It is worth noticing that the **modifies** clause of the method refers to a pointer. In this case the modified variable is an array rather then a single value. In order for proof obligations to consider all the modified memory locations we have to address every memory location of the array. The first attempt to prove determinism of the variables modification failed. We analyzed the cause of the failure and discovered that the postcondition does not specify the effect of the function on the values of the array at indexes exceeding value *count* The completed postcondition considers the whole range of memory locations addressed by the pointer *i*.

The proof obligations for RWFile class are concluded by theorems for member functions Write(char i) and Write(char * i, size_t count). The corresponding traits are shown in Figures 24 and 25 respectively. Protocols of LP proofs are documented in the Appendix, Figures 31 and 32 respectively.

```
class RWFile {
    typedef unsigned size_t;
    typedef char *String;

    uses File(RWFile for OpenFile,String for Name, String for
    MODE), Types(char);


public:
    RWFile(const char* filename, const char* mode=0)
    {
            modifies self;
            ensures if mode = 0 then
                        ∃f : File. of : OpenFile (if (f.name = filename ∧
                            of = open(f, READ_WRITE)) then
                                self' = of)
                        else
                        self' = open(create(filename, READ_WRITE),
                                READ_WRITE))
                    else
                    self' = open(create(filename, mode), mode);
    }
    ~ RWFile()
    {
            modifies self.File;
            ensures trashed(flush(self^));
    }
    long CurOffset()
    {
            ensures result = self^.fpointer;
    }
    RWBoolean Eof()
    {
            ensures result = (self^.fpointer = len(self^.data));
    }
    RWBoolean Erase()
    {
            modifies self;
            ensures self'.data = empty;
    }
    RWBoolean Error()
    {
            ensures result = error(self^);
```

```
}
RWBoolean Exists()
{
        ensures ∃file : File, name : Name, mode : MODE
        (result = (self^ = open(file, mode)∧
        file = create(name, READ_WRITE)));
}
RWBoolean Flush()
{
        modifies self.File;
        ensures result = ¬error(flush(self^));
}
const char* GetName()
{
        ensures result' = self^.file.name;
}
RWBoolean ISEmpty()
{
        ensures result = (self^.data = empty);
}
RWBoolean isValid() const
{
        ensures ∃f : File, m : MODE(result = (self^ = open(f, m)));
}
RWBoolean Read(char& c)
{
        requires len(self^.data) − self^.fpointer >= len(toByte(c^));
        modifies self.fpointer, c;
        ensures result = ¬error(self')∧
        toByte(c') = read(self^, len(toByte(c^)), self^.fpointer)∧
        self'.fpointer = self^.fpointer + len(toByte(c^));
}
RWBoolean Read(char* i, size_t count)
{
        requires len(self^.data) − self^.fpointer >=
        count * len(toByte((*i)^));
        modifies self.fpointer, *i;
        //Incomplete postcondition
        /*
        ensures result = ¬error(self')∧
        (∀ind : Int((ind >= 0∧
        ind <= count) => (toByte((*(i + ind))') =
        read(self^, len(toByte((*i)^)), self^.fpointer+
```

$ind * len(toByte((*i)\hat{}))).reddata) \wedge$
$self'.fpointer = self\hat{}.fpointer + count;$
*/
//Corrected postcondition
ensures $result = \neg error(self') \wedge$
$(\forall ind : Int($**if** $ind >= 0 \wedge$
$ind <= count$ **then** $toByte((*(i+ind))') =$
$read(self\hat{}, len(toByte((*i)\hat{})), self\hat{}.fpointer +$
$ind * len(toByte((*i)\hat{}))).reddata$
else $*(i+ind)' == *(i+ind)\hat{}) \wedge$
$self'.fpointer = self\hat{}.fpointer + count;$
}
RWBoolean SeekTo(long offset)
{
modifies $self.fpointer;$
ensures $result = (self'.fpointer = offset);$
}
RWBoolean SeekToBegin()
{
modifies $self.fpointer;$
ensures $result = (self'.fpointer = 1);$
}
RWBoolean SeekToEnd()
{
modifies $self.fpointer;$
ensures $result = (self'.fpointer = len(self\hat{}.data));$
}
RWBoolean Write(char i)
{
requires$\exists f : File\ (self\hat{} = open(f, WRITE) \vee$
$self\hat{} = open(f, READ\_WRITE));$
modifies $self;$
ensures $result = \neg error(self') \wedge$
$self' = write(self\hat{}, toByte(i), self\hat{}.fpointer);$
}
RWBoolean Write(char* i, size_t count)
{
requires $maxIndex(i) + 1 >= count \wedge$
$\exists f : File\ (self\hat{} = open(f, WRITE) \vee$
$self\hat{} = open(f, READ\_WRITE));$
modifies $self;$
ensures $result = \neg error(self') \wedge$
$\forall ind : Int(ind >= 0 \wedge ind <= count$

```
        ∧toByte((*(i + ind))^) =
        read(self', len(toByte((*i)^)), self^.fpointer+
        ind * len(toByte((*i)^)).reddata);
    }
    RWBoolean Exists(const char* filename )
    {
        ensures ∃f : File result = (filename = f.name∧
        file = create(filename, READ_WRITE));
    }
};
```

Figure 12: File class interface specification RWFile.lcc

```
File : trait
    includes Sequence(Byte, Data)
    File tuple of name : Name, data : Data, mode : MODE
    OpenFile tuple of file : File, data : Data, mode : MODE, fpointer : Int
    MODE enumeration of 0, READ, WRITE, READ_WRITE
    readeffect tuple of ofile : OpenFile, reddata : Data
    introduces
        create : Name, MODE → File
        open : File, MODE → OpenFile
        flush : OpenFile → OpenFile
        error : OpenFile → Bool
        read : OpenFile, Int, Int → readeffect
        write : OpenFile, Data, Int → OpenFile

    asserts
        ∀ f : File, opf, opf1 : OpenFile,
        mode, m : MODE, nm : Name, i, p : Int, dat : Data
            create(nm, m) == [nm, empty, m]
            open(f, READ) == [f, f.data, READ, 1]
            open(f, READ_WRITE) == [f, f.data, READ_WRITE, 1]
            open(f, WRITE) == [f, f.data, READ_WRITE, len(f.data)]
            % (flush(opf).file.data=opf.data;
            % after an incompleteness was detected
            %  the previous assertion was changed as follows
            flush(opf) = [[opf.file.name, opf.data, opf.file.mode], opf.data,
            opf.mode, opf.fpointer]
            read(opf, i, p).reddata == prefix(removePrefix(opf.data, p), i)
            read(opf, i, p).ofile == [opf.file, opf.data, opf.mode, p + i]
```

$write(opf, dat, p) == [opf.file, prefix(opf.data, p) \parallel dat \parallel$
$removePrefix(opf.data,$
$p + len(dat)), opf.mode, p + len(dat)]$

implies $\forall\ opf, opf_1 : OpenFile, dat : Data, i, p : Int$
$read(write(opf, dat, p), len(dat), p).reddata == dat$

converts *create*, *open*
exempting $\forall\ nm : Name,$
$f : File, dat : Data, p : Int write(open(f, READ), dat, p),$
$open(create(nm, READ), WRITE), open(create(nm, READ), READ\_WRITE),$
$create(nm, 0), open(f, 0)$

Figure 13: File.lsl trait

CompFileConst : **trait**
includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*),
Pointer(*Obj_character*, *character*, *String*)
introduces
$create' : String, String \rightarrow File$
$open' : File, String \rightarrow RWFile$
asserts
$\forall\ filename : String, mode : String, f : File, dat : Data, p : Int$
$create'(filename, mode) == [filename, empty, mode]$
$open'(f, READ) == [f, f.data, READ, 1]$
$open'(f, READ\_WRITE) == [f, f.data, READ\_WRITE, 1]$
$open'(f, WRITE) == [f, f.data, READ\_WRITE, len(f.data)]$
$write(open'(f, READ), dat, p) == write(open(f, READ), dat, p)$
$open'(create(filename, READ), WRITE) ==$
$open(create(filename, READ), WRITE)$
$open(create'(filename, READ), WRITE) ==$
$open(create(filename, READ), WRITE)$
$open'(create(filename, READ), READ\_WRITE) ==$
$open(create(filename, READ), READ\_WRITE)$
$open(create'(filename, READ), READ\_WRITE) ==$
$open(create(filename, READ), READ\_WRITE)$
$open'(f, 0) == open(f, 0)$
implies
$\forall\ filename : String, mode : String, f : File,$
$opf, opf_1, self, self_1, self', self1' : RWFile$
$open(f, mode) = open'(f, mode)$
$create(filename, mode) == create'(filename, mode)$

((**if** (*mode* = 0) **then**
(**if** (*f.name* = *filename*∧
opf = *open*(*f*, *READ_WRITE*)) **then** *self'* = *opf*
else
self' = *open*(*create*(*filename*, *READ_WRITE*), *READ_WRITE*))
else
self' = *open*(*create*(*filename*, *mode*), *mode*))∧
(**if** (*mode* = 0) **then**
(**if** (*f.name* = *filename*∧
opf = *open*(*f*, *READ_WRITE*)) **then** *self1'* = *opf*
else
self1' = *open*(*create*(*filename*, *READ_WRITE*), *READ_WRITE*))
else
self1' = *open*(*create*(*filename*, *mode*), *mode*))) ⇒ (*self'* = *self1'*)

Figure 14: Completeness proof obligation trait CompFileConst.lsl

CompFileDest : **trait**
includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for**
MODE), *Types*(*character*),
Pointer(*Obj_character*, *character*, *String*)
introduces
flush' : *RWFile* → *RWFile*
asserts
∀ *opf*, opf_1, *self*, $self_1$, *self'*, *self1'* : *RWFile*
flush'(*opf*) = [[*opf.file.name*, *opf.data*, *opf.file.mode*], *opf.data*,
opf.mode, *opf.fpointer*]
implies
∀ *opf*, opf_1, *self*, $self_1$, *self'*, *self1'* :
RWFile

flush(*self'*) = *flush'*(*self'*)
(*self'* = *flush*(*self*) ∧ *self1'* = *flush*(*self*)) ⇒ *self'* = *self1'*

Figure 15: Completeness proof obligations trait CompFileDest.lsl

CompFileError : **trait**
includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for**
MODE), *Types*(*character*),
Pointer(*Obj_character*, *character*, *String*)

introduces

$error' : RWFile \rightarrow Bool$

implies

$\forall\ opf : RWFile$

$error'(opf) = error(opf)$

Figure 16: Completeness proof obligations trait CompFileError.lsl

CompFileEOF : **trait**

includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*), *Pointer*(*Obj_character*, *character*, *String*)

introduces

$len' : Data \rightarrow Int$

asserts

$\forall\ p : Int, s_1, s_2, d, dat : Data,$

$e : Byte, t, t_1 : character, opf : RWFile, f : File$

$open(f, WRITE) == [f, f.data, READ\_WRITE, len'(f.data)]$

$write(opf, dat, p) == [opf.file, prefix(opf.data, p) \parallel dat \parallel removePrefix(opf.data, p + len'(dat)), opf.mode, p + len'(dat)]$

$isPrefix(s_1, s_2) == s_1 = prefix(s_2, len'(s_1))$

$len'(empty) == 0$

$len'(d \vdash e) == len'(d) + 1$

$len'(toByte(t)) = len'(toByte(t_1))$

implies

$\forall\ q : Data$

$len'(q) = len(q)$

Figure 17: Completeness proof obligations trait CompFileEOF.lsl

CompFileExists : **trait**

includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*), *Pointer*(*Obj_character*, *character*, *String*)

introduces

$create' : String, String \rightarrow File$

```
    open' : File, String → RWFile
  asserts
    ∀ filename : String, mode : String, f : File, dat : Data, p : Int

      create'(filename, mode) == [filename, empty, mode]
      open'(f, READ) == [f, f.data, READ, 1]
      open'(f, READ_WRITE) == [f, f.data, READ_WRITE, 1]
      open'(f, WRITE) == [f, f.data, READ_WRITE, len(f.data)]
      write(open'(f, READ), dat, p) ==
      write(open(f, READ), dat, p)
      open'(create(filename, READ), WRITE) ==
      open(create(filename, READ), WRITE)
      open(create'(filename, READ), WRITE) ==
      open(create(filename, READ), WRITE)
      open'(create(filename, READ), READ_WRITE) ==
      open(create(filename, READ), READ_WRITE)
      open(create'(filename, READ), READ_WRITE) ==
      open(create(filename, READ), READ_WRITE)
  implies
    ∀ filename : String, mode : String, f : File,
      opf, opf_1, self, self_1, self', self1' : RWFile
      open(f, mode) = open'(f, mode)
      create(filename, mode) == create'(filename, mode)
```

Figure 18: Completeness proof obligations trait CompFileExists.lsl

```
CompFileFlush : trait
  includes File(RWFile for OpenFile, String for Name, String for
    MODE), Types(character),
    Pointer(Obj_character, character, String)
  introduces
    flush' : RWFile → RWFile
  asserts
    ∀ opf : RWFile
      flush(opf) = [[opf.file.name, opf.data, opf.file.mode], opf.data,
      opf.mode, opf.fpointer]
  implies
    ∀ opf : RWFile
      flush'(opf) == flush(opf)
```

Figure 19: Completeness proof obligations trait CompFileFlush.lsl

```
CompFileisValid : trait
```

includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*),
Pointer(*Obj_character*, *character*, *String*)
introduces
open' : *File*, *String* → *RWFile*
asserts
∀ *filename* : *String*, *mode* : *String*, *f* : *File*, *dat* : *Data*, *p* : *Int*
open'(*f*, *READ*) == [*f*, *f.data*, *READ*, 1]
open'(*f*, *READ_WRITE*) == [*f*, *f.data*, *READ_WRITE*, 1]
open'(*f*, *WRITE*) == [*f*, *f.data*, *READ_WRITE*, *len*(*f.data*)]
write(*open'*(*f*, *READ*), *dat*, *p*) == *write*(*open*(*f*, *READ*), *dat*, *p*)
open'(*create*(*filename*, *READ*), *WRITE*) ==
open(*create*(*filename*, *READ*), *WRITE*)
open'(*create*(*filename*, *READ*), *READ_WRITE*) ==
open(*create*(*filename*, *READ*), *READ_WRITE*)
implies
∀ *mode* : *String*, *f* : *File*
open(*f*, *mode*) = *open'*(*f*, *mode*)

Figure 20: Completeness proof obligations trait CompFileisValid.lsl

CompFileDest : **trait**
includes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*),
Pointer(*Obj_character*, *character*, *String*)
introduces
flush' : *RWFile* → *RWFile*
asserts
∀ *opf*, opf_1, *self*, $self_1$, *self'*, *self1'* : *RWFile*
flush'(*opf*) = [[*opf.file.name*, *opf.data*, *opf.file.mode*], *opf.data*,
opf.mode, *opf.fpointer*]
implies
∀ *opf*, opf_1, *self*, $self_1$, *self'*, *self1'* :
RWFile

flush(*self'*) = *flush'*(*self'*)
(*self'* = *flush*(*self*) ∧ *self1'* = *flush*(*self*)) ⇒ *self'* = *self1'*

Figure 21: Completeness proof obligations trait CompFileDest.lsl

$CompFileRead_1$: **trait**

assumes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String*.**for** *MODE*), *Types*(*character*), *Pointer*(*Obj_character*, *character*, *String*)

introduces

% requires

$len' : Data \rightarrow Int$

$toByte' : character \rightarrow Data$

% ensures

$read' : RWFile, Int, Int \rightarrow readeffect$

asserts

character **partitioned by** *toByte'*

$\forall$

$s_1, s_2, d, q : Data, e, e_1 : Byte, t, t_1 : character, dat : Data, self, opf : RWFile, p, i : Int, f : File$

% requires

$open(f, WRITE) == [f, f.data, READ\_WRITE, len'(f.data)]$

$read'(opf, i, p).reddata == prefix(removePrefix(opf.data, p), i)$

$read'(opf, i, p).ofile == [opf.file, opf.data, opf.mode, p + i]$

$write(opf, dat, p) == [opf.file, prefix(opf.data, p) \parallel dat \parallel removePrefix(opf.data, p + len'(dat)), opf.mode, p + len'(dat)]$

$isPrefix(s_1, s_2) == s_1 = prefix(s_2, len'(s_1))$

$len'(empty) == 0$

$len'(d \vdash e) == len'(d) + 1$

$toType(toByte'(t)) == t$

$toByte'(toType(dat)) == dat$

$len'(toByte'(t)) = len'(toByte'(t_1))$

$((len(self.data) - self.fpointer) \geq len(toByte(t))) \Rightarrow (read'(self, i, p) = read(self, i, p))$

implies

$\forall$

$s_1, s_2, d, q : Data, e, e_1 : Byte, t, t_1, t', t1' : character, dat : Data, self, self', opf : RWFile, p, i : Int$

$(read'(self, i, p) = read(self, i, p))$

$toByte'(t) = toByte(t)$

$len'(q) = len(q)$

% determinism

```
(
(toByte(t') = read(self, len(toByte(t)), self.fpointer).reddata∧
self'.fpointer = self.fpointer + len(toByte(t))
)
∧
(toByte(t1') = read(self, len(toByte(t)), self.fpointer).reddata∧
self'.fpointer = self.fpointer + len(toByte(t))
)
) ⇒
(t' = t1')
```

Figure 22: Completeness proof obligations trait CompFileRead1.lsl

```
CompFileRead2 : trait
    assumes File(RWFile for OpenFile, String for Name, String for
        MODE), Types(character),
        Pointer(character, character, String)
    introduces
        % requires
        len' : Data → Int
        toByte' : character → Data
        % ensures
        read' : RWFile, Int, Int → readeffect
    asserts
        character partitioned by toByte'
        ∀
            s1, s2, d, q : Data, e, e1 : Byte, t, t1 : character, dat : Data, self, opf :
RWFile, p, i, count : Int, f : File,
            ptr : String
            % requires
            open(f, WRITE) == [f, f.data, READ_WRITE,
                len'(f.data)]
            read'(opf, i, p).reddata ==
                prefix(removePrefix(opf.data, p), i)
            read'(opf, i, p).ofile == [opf.file,
                opf.data, opf.mode, p + i]
            write(opf, dat, p) == [opf.file,
                prefix(opf.data, p) || dat || removePrefix(opf.data,
                p + len'(dat)), opf.mode, p + len'(dat)]
            isPrefix(s1, s2) == s1 = prefix(s2, len'(s1))
            len'(empty) == 0
            len'(d ⊢ e) == len'(d) + 1
```

$toType(toByte'(t)) == t$
$toByte'(toType(dat)) == dat$
$len'(toByte'(t)) = len'(toByte'(t_1))$

$((len(self.data) - self.fpointer) \geq (count * len(toByte(*ptr)))) \Rightarrow$
$(read'(self, i, p) = read(self, i, p))$

implies

$\forall$

$s_1, s_2, d, q : Data, e, e_1 : Byte, t, t_1, t', t1' : character, dat : Data, self, self', self_1,$
$self1', opf : RWFile, p, j, count, ind : Int, i, i_1, i', i1', ptr,$
$ptr_1 : String$
$(read'(self, j, p) = read(self, j, p))$
$toByte'(t) = toByte(t)$
$len'(q) = len(q)$
% determinism
(
((**if** $ind \geq 0 \wedge ind \leq count$ **then** $toByte(*(i + ind)) =$
$read(self, len(toByte(*i)), self.fpointer + (ind * len(toByte(*i)))).reddata$
else $*(i + ind) = *(i + ind)) \wedge self'.fpointer = self.fpointer + count) \wedge$
((**if** $ind \geq 0 \wedge ind \leq count$ **then** $toByte(*(i + ind)) =$
$read(self, len(toByte(*i)), self.fpointer + (ind * len(toByte(*i)))).reddata$
else $*(i + ind) = *(i + ind)) \wedge self1'.fpointer = self.fpointer + count)$
$) \Rightarrow self'.fpointer = self1'.fpointer$

(
((**if** $ind \geq 0 \wedge ind \leq count$ **then** $toByte(*(i + ind)) =$
$read(self, len(toByte(*i)), self.fpointer + (ind * len(toByte(*i)))).reddata$
else $*(i + ind) = *(i + ind)) \wedge self'.fpointer = self.fpointer + count) \wedge$
((**if** $ind \geq 0 \wedge ind \leq count$ **then** $toByte(*(i_1 + ind)) =$
$read(self, len(toByte(*i)), self.fpointer + (ind * len(toByte(*i)))).reddata$
else $*(i_1 + ind) = *(i + ind)) \wedge self'.fpointer = self.fpointer + count)$
$) \Rightarrow (*(i + ind) = *(i_1 + ind))$

Figure 23: Completeness proof obligations trait CompFileRead2.lsl

CompFileWrite$_1$: **trait**

assumes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*), *Pointer*(*character*, *character*, *String*)

introduces

% requires

$open' : File, String \rightarrow RWFile$

$write' : RWFile, Data, Int \rightarrow RWFile$

% ensures

$write' : RWFile, Int, Int \rightarrow RWFile$

$toByte' : character \rightarrow Data$

asserts

$\forall\ i, j, n : Int, p : String,$

$f : File, self, opf, opf_1 : RWFile, mode, m : String, nm : String, dat : Data,$

$t, t_1 : character,$

$c : Data$

% requires

$open'(f, READ) == [f, f.data, READ, 1]$

$open'(f, READ\_WRITE) == [f, f.data, READ\_WRITE, 1]$

$open'(f, WRITE) == [f, f.data, READ\_WRITE, len(f.data)]$

$open'(create(nm, READ), WRITE) == open(create(nm, READ), WRITE)$

$open'(create(nm, READ), READ\_WRITE) == open(create(nm, READ), READ\_WRITE)$

% ensures

$\neg(self = open(f, WRITE) \vee self = open(f, READ\_WRITE)) \Rightarrow write(self, c, i) = write(self, c, i)$

$\neg(self = open(f, WRITE) \vee self = open(f, READ\_WRITE)) \Rightarrow toByte'(t) = toByte(t)$

$write'(opf, dat, i) == [opf.file, prefix(opf.data, i) \parallel dat \parallel removePrefix(opf.data, i + len(dat)), opf.mode, i + len(dat)]$

$toType(toByte'(t)) == t$

$toByte'(toType(dat)) == dat$

$len(toByte'(t)) = len(toByte'(t_1))$

implies

$\forall\ p, i, i', j : String, f : File, m : String, c : Data, opf : RWFile,$

$t : character,$

$ind, count : Int, self, self', self1' : RWFile, result : Bool$

% requires

$open'(f, m) == open(f, m)$
% ensures
$write'(opf, c, count) = write(opf, c, count)$
$toByte(t) = toByte(t)$
% determinism
$((self' = write(self, toByte(*i)),$
$self.fpointer) \wedge result)$
$\wedge$
$(self1' = write(self, toByte(*i)),$
$self.fpointer) \wedge result)) \Rightarrow$
$(self' = self1')$

Figure 24: Completeness proof obligations trait CompFileWrite1.lsl

CompFileWrite$_2$: **trait**
% assumes File, Types(Obj_S), Queue(E, C), Pointer(Obj_S,S,Ptr)
assumes *File*(*RWFile* **for** *OpenFile*, *String* **for** *Name*, *String* **for** *MODE*), *Types*(*character*), *Pointer*(*character*, *character*, *String*)
introduces
% requires
$maxIndex' : String \rightarrow Int$
$open' : File, String \rightarrow RWFile$

% ensures
$read' : RWFile, Int, Int \rightarrow readeffect$
$len' : Data \rightarrow Int$
$toByte' : character \rightarrow Data$
asserts
$\forall\ i, j, n : Int, p : String,$
$f : File, self, opf, opf_1 : RWFile, mode, m : String,$
$nm : String, dat : Data, t, t_1 : character,$
$c, d : Data, e : Byte$

% requires
$maxIndex'(p) == maxIndex(p.locs) - p.idx$
$legalIndex(p, i) == (minIndex(p) \leq i) \wedge (i \leq maxIndex'(p))$
% ensures

$\neg(self = open(f, WRITE) \vee self = open(f, READ\_WRITE)) \Rightarrow (read'(self, i, j) = read(self, i, j))$

$\neg(self = open(f, WRITE) \vee self = open(f, READ\_WRITE)) \Rightarrow (toByte'(t) = toByte(t))$

$\neg(self = open(f, WRITE) \vee self = open(f, READ\_WRITE)) \Rightarrow (len(c) = len(c))$

$open'(f, READ) == [f, f.data, READ, 1]$
$open'(f, READ\_WRITE) == [f, f.data, READ\_WRITE, 1]$
$open'(f, WRITE) == [f, f.data, READ\_WRITE, len(f.data)]$
$open'(create(nm, READ), WRITE) == open(create(nm, READ), WRITE)$
$open'(create(nm, READ), READ\_WRITE) == open(create(nm, READ), READ\_WRITE)$
$open'(f, 0) == open(f, 0)$
$read'(opf, i, j).reddata == prefix(removePrefix(opf.data, j), i)$
$read'(opf, i, j).ofile == [opf.file, opf.data, opf.mode, j + i]$
$len'(empty) == 0$
$len'(d \vdash e) == len'(d) + 1$
$toType(toByte'(t)) == t$
$toByte'(toType(dat)) == dat$
$len'(toByte'(t)) = len'(toByte'(t_1))$

implies

$\forall\ p, i, i', j : String, f : File, m : String, c : Data, opf : RWFile, t : character,$
$ind, count : Int, self, self', self1' : RWFile, result : Bool$
% requires
$maxIndex'(p) == maxIndex(p)$
$open'(f, m) == open(f, m)$
% ensures
$read'(opf, ind, count) = read(opf, ind, count)$
$len'(c) = len(c)$
$toByte(t) = toByte(t)$
% determinism
$((ind \geq 0 \wedge ind \leq count \wedge$
$toByte(*(i + ind)) = read(self', len(toByte(*i)),$
$self.fpointer + (ind * len(toByte(*i)))).reddata \wedge result)$
$\wedge$

$(ind \geq 0 \wedge ind \leq count \wedge$.
$toByte(*(i + ind)) = read(self1', len(toByte(*i)),$
$self.fpointer + (ind * len(toByte(*i)))).reddata \wedge result)) \Rightarrow$
$(self' = self1')$

Figure 25: Completeness proof obligations trait CompFileWrite2.lsl

6.2 Semi-decidability of the first order logic theories and incompleteness

First order logic is semi-decidable We can prove that the first order logic statement is indeed a consequence of a given theory. In general, we can not prove that some statement is not among the consequences of a given theory.

In the light of the completeness check methodology, addressed in this thesis, incompleteness of specification can not be proved due to the semi-decidability of the first order logic. Nevertheless, the methodology provides algorithmic steps to prove the completeness of Larch/C++ interface specifications. Unsuccessful attempts to prove completeness of a specification should resort to the analysis of the proof to conclude the incompleteness, localize it and rectify the problem.

6.2.1 Incompleteness detection and localization strategy

The common strategies of proving theorems using Larch Prover can be applied to prove completeness [GG93]. On the other hand, there are only certain kinds of first order logic formulas used in the completeness verification theorems. As discussed in the previous chapter, we experience particular proof methods more often than others. In addition, LP provides excessive information about the proof status. These allow us to identify and localize the cause of a proof failure. Unsuccessful attempts to prove a conjecture lead to the assumption that the conjecture is not a theorem.

If the conjecture was intended to prove totality of an LSL function, then we have to analyze the corresponding assertions in the LSL theory. Such analysis usually

identifies either a trace for proving the conjecture or incompleteness of the assertions stated. In the latter case an incompleteness is localized to the particular LSL term and we have to complete the set of equations stating the term behavior.

If we failed to prove a conjecture concerning determinism of a variable modification by the class member-function, then the corresponding postcondition must be examined. Thus, again, we know the location of a possible incompleteness. The analysis of the postcondition formula usually provides the trace to prove the conjecture or identifies that there are cases when the outcome of the formula is not defined. For example, if a postcondition formula is:
$A => B(v)$, where v is a variable modified by the function, then the correction would be to complete the formula with the case when A is $FALSE$. Possible solution is to modify the formula:
if A **then** $B(var)$ **else** C.

6.2.2 Completeness evaluation of abstract base classes

Next, we consider pure abstract base classes. An abstract base class is a class that has only pure virtual functions and, therefore, can not be instantiated into C++ objects. By its nature, an abstract base class can have only partially specified behavior. Indeed, it consists of the virtual functions which are to be bound to the functions in the subclasses, having completely defined behavior. As a result, abstract base classes do not need to have a completely specified behavior. Virtual function must have complete specification only in the classes where the actual function call is to be resolved. If this is the case, the virtual function must undergo the same completeness verification as any other non virtual function.

Private functions cannot be used outside the class where they are defined. Therefore, it does not make sense to include them into specification aimed to facilitate black box reuse. Public, friends, and protected functions can be used by other classes, though there are some limitations on such a use for protected and friends functions (former can be called by subclasses of the class and the latter can be called by specially declared friend classes). Thus, specifications of these functions must satisfy the completeness criteria outlined in this thesis.

6.2.3 Incompleteness of exceptional conditions in specification

Some classes have functions to determine whether an exception has occured since the object was created. For example, if an object is of a container type, there could be an exception condition when a function inserting an element into the container is called and system runs out of memory. Exception is an event incurred by the environmental circumstances rather than in agreement with the logic of a class behavior (that is why it is an exception and not a class behavioral feature). Therefore, the nature of exceptions implies incompleteness of their specification in a framework of the class specified.

6.2.4 Completeness methodology and inheritance

C++ class can have an inheritance tree of its base classes. If this is the case, completeness of the class can be checked recursively :

- Functions redefined in the class interface specification must have complete specification. That is, the completeness criteria must be satisfied for specifications of the subset of member-functions, consisting of virtual functions redefined in the class.
- The functions that are newly defined in the class, must satisfy the completeness verification unless the class is an abstract one.
- Non abstract base classes must have complete specification.

Notice, that this approach does not demand pure virtual functions to have complete specification.

If class B is a subtype of class A and B is complete then A is also complete. No separate verification of completeness is necessary. If B is a general inherited subclass of A (not subtype inheritance), then it is sufficient to apply the completeness verification methodology to those functions of A that do not conform to subtype property.

Chapter 7

Completeness Verification Tool

In previous chapters we developed a methodology to verify the completeness of formal specification intended for a black-box reuse and identified the algorithm to apply the methodology semi-automatically with the help of Larch Theorem Prover. Analysis of the possible incompleteness in specification based on the result of applying Completeness Verification Algorithm was addressed as well. In this chapter we consider features and design issues of a tool which would provide convenient environment to work with Larch/C++ specifications, facilitating their completeness verification process.

7.1 Feature analysis

The process of completeness verification can be viewed as three consecutive activities:

1. A number of the first order logic theories with theorems are prepared based on the Larch/C++ formal specification.

2. Each of the theories serves as an input for Larch Theorem Prover with the goal to prove the theorems stated.

3. An incompleteness identified in the second step, if any, is analyzed and necessary corrections to the specification are made to remove an incompleteness detected.

The first step takes as an input a Larch/C++ specification, processes it and produces LSL traits with theorems. This activity can be automated completely based

on the theory developed in this thesis which then evolved into the Completeness Verification Algorithm. The algorithm is computationally feasible. It is applied to build LSL traits with theorems after LSL signatures are parsed from the expressions in the member-functions of the interface specification. The LSL traits built by the algorithm will be a source for the input into the Larch Theorem Prover in order to prove conjectures stated in the traits.

Once LSL traits are obtained, they have to be converted to the LP format, and user will guide LP through the theorem proving. Theorem proving itself can be only semi-automated. Since there are as many LP inputs as many functions are in the analyzed specification, the task of the tool would be to sense when all the theorems of the current trait have been proven and to retrieve the next trait, clean-up LP and restart it with the new input. In addition, completeness must be concluded when all the LP inputs are successfully processed. Besides, the tool has to have an interface to interact with a user and display traits to be input into LP, so that user can analyze them and possibly process only a subset of all the traits or only a subset of theorems. Such selectiveness is useful because generally, there are many identical theorems generated from the different member-functions of a specification and excluding repetitions would reduce the amount of work required to verify the completeness. Unfortunately, automated analysis of the theorem repetitions can not be achieved due to the fact that an identity of the theories where respective theorems appear has to be established as well and the latter is a semi-decidable task, involving the comparison of the consequence closures of two theories and logical analysis versa algorithmic steps.

If a theorem proving attempt is not successful, the tool should classify the problem as discussed in 6.2.1. Thus, it will give a hint to the user with suggestions on location(s) of a possible incompleteness. The tool will provide popped-up multiple window editor loaded with the interface functions or traits which caused a possible incompleteness. After the editor was used to make corrections, the tool restarts LP with the corrected input.

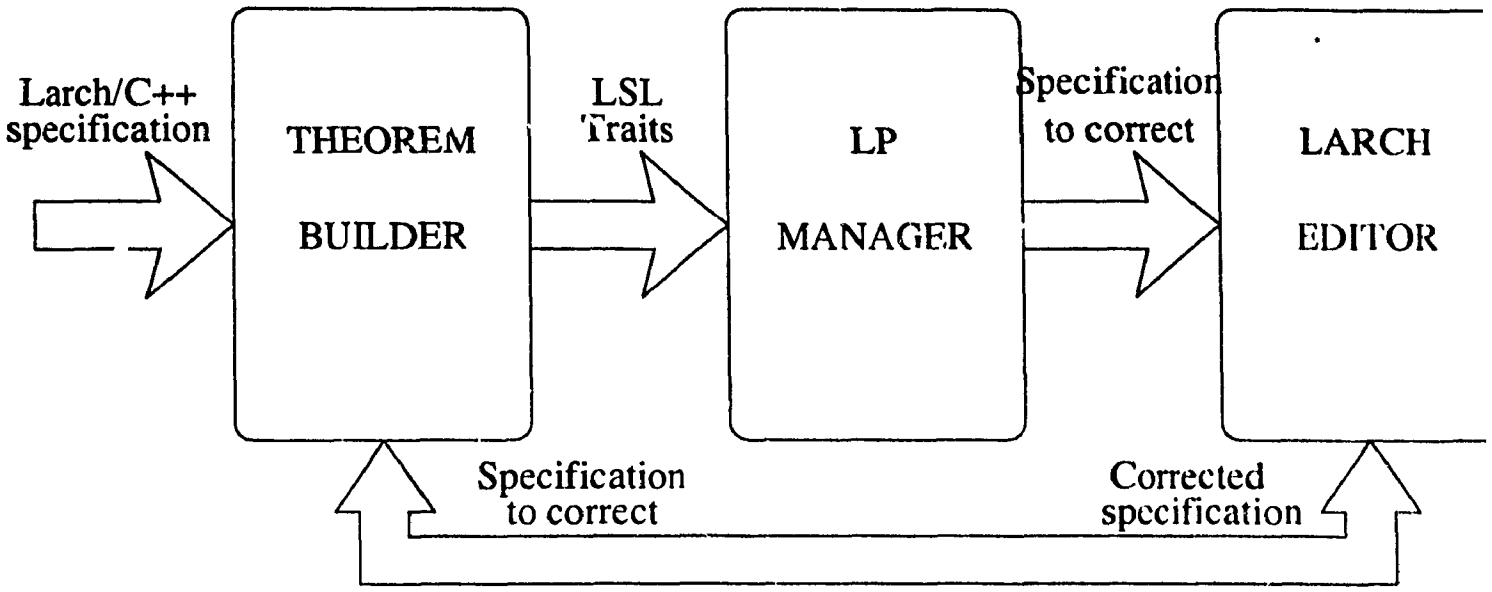

Figure 26: Completeness Verification Tool

7.2 Subsystem decomposition

We decompose the Completeness Verification Tool architecture into subsystems based on the feature analysis presented in the previous section. The subsystems are shown in Figure 26.

Theorem builder subsystem is responsible for constructing LSL traits containing theorems from a given Larch/C++ specification. LSL traits and theorems are built as identified in the Completeness verification algorithm presented in 4.2. Larch/C++ specification must undergo syntactic and type checking prior to being analyzed for completeness. The former activities are incorporated in the Theorem Builder subsystem as well. This means that Theorem Builder subsystem uses Larch/C++ syntax checker and type checker. X-Motif interface can be built to provide a graphical communication with the user. Interactions with the user are required to inform that LP input traits are ready so that the user can direct the system to start LP to process the traits. If errors are detected during syntax or type checking in the specification then the user is informed and may activate editing facilities of the system to correct detected errors. Editing of Larch/C++ specifications is provided by the Larch Editor

subsystem.

LP Manager subsystem manages LP processing of the theorems. This subsystem responsibilities include:

- Produce LP files from the LSL trait output by the Trait Builder subsystem. This might be done using LP file generating facilities which are included into the standard LP package.
- Start Larch Theorem Prover.
- Clean-up LP and load next LP input file when the current LP file processing has been completed.
- Interact with the user to inform him on the proof status. This includes user notification when all the conjectures from the current file have been proven as well as interrogating whether to continue with the next LP input file. User directives are required when a proof attempt is not successful as well:
  1. Abort processing of the current file.
  2. Edit trait(s) and Larch/C++ member-functions, involved in the current theorem when user decides that analysis and possibly correction of the specification is required. **Larch Editor** subsystem services will be used in this case.
  3. Notify user when all of the LP input files have been processed successfully and, therefore, the specification is proven to be complete.

Larch Editor subsystem provides convenient editing environment for Larch/C++ specifications. This includes the ability to identify member function or/and LSL trait(s) for a given LP input file and a theorem in it. This will provide convenience to the user when LP manager subsystem outputs an LP file with a conjecture for which proof attempt(s) were not successful. Incompleteness localization strategy identified in 6.2.1 of this thesis is used to analyze the cause of a proof failure. This analysis might lead to localization of the possible incompleteness to one of the following:

- Formula(s) in the interface specification
- Term signature in the LSL trait

There must be some meta-data available to map a given LP file and conjecture in it into the member function and LSL trait(s) involved into producing of the conjecture. Larch Editor subsystem is used also when syntactic or type mismatch problems were detected by the Trait Builder subsystem and the file needs to be edited to make the necessary corrections. The subsystem's user interface should provide communication with the user to hint at possible location of incompleteness. to manage specification files, to pass corrected specification to the Theorem Builder subsystem when file editing is completed.

Chapter 8

Conclusion

8.1 Summary

Object-Oriented class reuse can be done in a black box fashion only if a reuser is given precise information about the interface behavior of a class. In order to express the behavior of a class the trace assertion approach was used. We identified that the appropriate vehicle to deliver the knowledge of a class behavior is its formal specification. As such, formal specification must be complete to provide the intended information in a full, exhaustive manner. Since the prime concern of this thesis is software written in C++ programming language, we used Larch/C++ formal specification language to write specifications for C++ class interfaces.

Though Larch/C++ was used before as a research level specification language, no work has been done to specify commercial C++ libraries. We pioneered this work by writing specification for classes from Rogue Wave C++ library. During the course of this work experience of writing Larch/C++ specifications was gained. We identified many general solutions and methods for specifying C++ specific aspects. The specifications written by our research group are summarized in the report [ACCUA94] submitted to Bell Northern Research.

In this thesis we focused on studying the completeness of formal specifications. We identified that general definition of completeness can not be applied in practice due to the semi-decidability of the first-order logic. Instead, we accepted an ad hoc completeness criteria which suits most of the specifications of C++ classes intended

for a black-box reuse. The core of this thesis is devoted to finding a solution to verify completeness of Larch/C++ specifications.

Because our aim was to develop a feasible algorithm for completeness verification, we derived sufficient conditions for a specification to be complete, which are bound to the properties of the first order logic theories. We proved sufficiency of the stated conditions to ensure that the truth of the initial completeness criteria is implied. These sufficient conditions formed the Completeness Verification Methodology. Though the Completeness Verification Methodology identified what has to be done to ensure completeness, it does not state how to achieve these goals.

Each of the items in the Completeness Verification Methodology was analyzed in order to find practical solutions to achieve completeness verification for Larch/C++ specifications. These solutions were successfully found and summarized into the Completeness Verification Algorithm. Completeness Verification Algorithm provides steps to form first order logic assertions and theorems. The latter must be proved to conclude completeness of the specification.

Larch Theorem Prover, an automatic proof assistant developed for Larch Shared Language was identified as a suitable tool for proving theorems stated in Larch traits. We provided detailed examples of forming completeness verification theories and proving theorems stated there using LP. We also identified guidelines to prove methods that are to be applied to carry out the completeness verification. We identified how to localize incompleteness detected as a result of an unsuccessful proof attempt, followed by the analysis of the cases of incompleteness and solutions for making specification complete as well as the cases when completeness can not be achieved. Finally, this thesis provides the algorithm for completeness verification of specification of C++ classes, having an inheritance tree as well as classes containing virtual functions and exceptions.

8.2 Future work

Larch/C++ specification language is not a finished software tool. Though its major constructs are stable, designers of the language keep making changes to the syntax

and semantics of Larch/C++, increasing its expressive power and making it to be more suitable for specifying sophisticated structure and behavior of C++ objects. Completeness verification algorithm may require some adjustments and maintenance while new Larch/C++ features are introduced. For example, a present consideration of the authors of Larch/C++ is to allow multiple **requires-ensures** clause within the same member-function to facilitate sub-typing of inheritance checking. Should this construct be accepted, a slight modification of the Completeness Verification Algorithm is needed. That is, instead of considering every member-function specification, every pair of **requires-ensures** clause must be considered. Nevertheless, the theoretical value of the Completeness Verification Methodology and Algorithm will not be decreased because the core algorithm of the developed in this thesis is a general solution for a family of model-oriented specifications rather than for any particular language feature or even a particular specification language.

The approach identified in this thesis can be adapted for the development of a completeness verification methodology for other module specification languages, for instance, VDM and Z. The Completeness Verification Methodology is mostly to be valid for these languages as stated for Larch/C++. The Completeness Verification Algorithm will be very similar to that for Larch/C++ except that it is likely to be simpler due to the use of the predefined types in VDM and Z. Another reason to expect a more simple completeness verification is the fact that Z and VDM do not have object-oriented features.

Theory provided by this thesis is a foundation for a tool which will provide convenient environment for Larch/C++ specification development and its completeness verification. Since the area of application of formal specifications keeps growing, the tool could incorporate more features than is outlined in this thesis. There could be facilities for using formal specifications for module's testing and using theorem prover to establish some other behavioral properties of a specification, not just its completeness.

It is important that the tool built from this methodology is subjected to an extensive testing. The plan is to use the Larch/C++ specifications currently being

developed by the new members of our research group as a testbed for the completeness verification tool. Moreover, the completeness verification tool and the black-box testing tool [Cel95] will be combined and tried in an industrial reuse environment of C++ class library[rogue93].

Another long term goal here is to investigate the applicability and effectiveness of our work in the context of reuse of C++ frameworks.

Bibliography

[AK94] V.S. Alagar and D. Kourkopolous *(In)completeness in specifications*, *Information and Software technology*, 1994, **36** (6) 331-342.

[ACCUA94] V.S. Alagar, P. Colagrosso, A. Celer, I. Umansky, R. Achuthan *Formal Specifications for Effective Black-Box Reuse*, Phase I Progress Report, September 1994.

[Boo86] Booch, G. Object-Oriented Development, *IEEE Transactions on Software Engineering*, SE-12, 2, pp. 211-221, 1986.

[BP86] W. Bartussek and D.L. Parnas, *Using assertions about trace to write abstract specifications for software modules* In Gehani and McGettrick, editors, *Software Specification Techniques*, Addison Wesley, 1986.

[Cel95] A. B. 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.

[CH87] Cohen, B. Harwood, W. T. and Jackson, M. I. *The Specification of Complex Systems*, Addison-Wesley, 1987.

[CAA93] P. Colagrosso, R. Achuthan, V.S. Alagar, *Evaluating the completeness of C++ class interface specifications for software reuse*, Technical Report, Department of Computer Science, Concordia University, Montreal, Canada, 1993.

[Col93] P. Collagrosso, *Formal specification of C++ class interface for software reuse*, Master of Computer Science Thesis, Department of Computer Science, Concordia University, Montreal, Canada, April 1993.

[GH78] J.V. Guttag & J.J. Horning, *The algebraic specification of abstract data types*, Acta Infor., vol. 10, pp.27-52, 1978

[GG93] Stephen J. Garland and John V. Guttag, A Guide to LP, The Larch Prover, Massachusetts Institute of Technology, March 1993.

[Kai87] Kaizhi Yue, *What Does It Mean To Say That A Specification Is Complete*, IEEE, 1987.

[Leava94] G. Leavens, *Larch: Handbook*, Springer-Verlag, 1993.

[Mc68] .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, Brussels, Belgium, 1968, pp. 138-150.

[Parn93] D.L. Parnas, *Predicate Logic for Software Engineering*, IEEE Transactions on Software Engineering, vol.19, September 1993.

[PaW89] D.L. Parnas & Y. Wang, *The Trace Assertion Method of Module Interface Specification*, *TR 89-261*, Telecommunications Research Institute of Ontario (TRIO), Queen's University, 1989.

[PW93] D.L. Parnas & Y. Wang, *Simulating the Behaviour of Software Modules by Trace Rewriting*, IEEE 15th International Conference on SE, 1993.

[rogue93] Rogue Wave, *Tools.h++ Class Library v6.0*, Rogue Wave Software, 1993.

[Strous94] B. Stroustrup, *The C++ Programming Language*, AT&T Bell Laboratories Murray Hill, New Jersey, 1994.

[Wing90] Jeannette M. Wing, *A Specifier's Introduction to Formal Methods*, IEEE Computer, 1990.

Appendices

Proofs for completeness verifications of RWFile C++ class from Rogue Wave class library

```
Larch Prover (24 January 1994) logging on 20 November 1994 12:38:54 to
'/mnt/kbs1/BBRS/uman/thesis/lp/file/CompFileConst lplog'.

LP2:  declare variables
Please enter variable declarations, terminated with a '..' line, or '?' for
help:
  filename: String
  mode  String
  f: File
  opf: RWFile
  opf1  RWFile
  self: RWFile
  self1  RWFile
  self': RWFile
  self1'. RWFile
  .

LP5: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help.
  open(f, mode) = open'(f, mode)
  ..


The current conjecture is StringTheorem 4.

The equations cannot be ordered using the current ordering.

Conjecture StringTheorem 4  open(f, mode) == open'(f, mode)
Proof suspended

LP7: display induction File

Induction rules:

File.1:  File generated by __mixfix1
File.9:  RWFile generated by __mixfix2
File.19: String generated by 0:->String, READ:->String, WRITE:->String,
           READ_WRITE
File.29: readeffect generated by __mixfix3

LP8: resume by induction on mode using File.19

Conjecture StringTheorem.4: Subgoals for proof by induction on 'mode'
Basis subgoals:
  StringTheorem.4.1: open(f, 0) == open'(f, 0)
  StringTheorem.4.2: open(f, READ) == open'(f, READ)
  StringTheorem.4.3: open(f, WRITE) == open'(f, WRITE)
  StringTheorem.4.4: open(f, READ_WRITE) == open'(f, READ_WRITE)
The induction step is vacuous.


The current conjecture is subgoal StringTheorem.4.1.
```

```
Subgoal StringTheorem.4.1: open(f, 0) == open'(f, 0)
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.4.2.

Subgoal StringTheorem.4.2: open(f, READ) == open'(f, READ)
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.4.3.

Subgoal StringTheorem.4.3. open(f, WRITE) == open'(f, WRITE)
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.4 4.

Subgoal StringTheorem.4 4: open(f, READ_WRITE) == open'(f, READ_WRITE)
[] Proved by normalization.

The current conjecture is StringTheorem 4.

Conjecture StringTheorem 4. open(f, mode) == open'(f, mode)
[] Proved by induction on 'mode'.

LP9· prove
Please enter a conjecture to prove, terminated with a '..' line, or '?  for
help
  create(filename, mode) == create'(filename, mode)
  .
The current conjecture is StringTheorem.5

Conjecture StringTheorem 5. create(filename, mode) == create'(filename, mode)
[] Proved by normalization.

LP34. prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help
  ((if(mode = 0·String, if(__sel7_name(f) = filename & opf = open(f,
    READ_WRITE), self' = opf, self' = open(create(filename, READ_WRITE),
    READ_WRITE)), self' = open(create(filename, mode), mode))) & (if(mode =
    0.String, if(__sel7_name(f) = filename & opf = open(f, READ_WRITE), self1'
    = opf, self1' = open(create(filename, READ_WRITE), READ_WRITE)), self1' =
    open(create(filename, mode), mode)))) => self' = self1'
  .


The current conjecture is StringTheorem.8.

The equations cannot be ordered using the current ordering.

Conjecture StringTheorem.8:
  (if(0 = mode,
      if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
         opf = self',
         open(create(filename, READ_WRITE), READ_WRITE) = self'),
      open(create(filename, mode), mode) = self')
    & if(0 = mode,
         if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
            opf = self1',
            open(create(filename, READ_WRITE), READ_WRITE) = self1'),
         open(create(filename, mode), mode) = self1'))
   => (self' = self1')
```

```
   == true
 Current subgoal:
   (if(0 = mode,
       if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
            & (__sel7_name(f) = filename),
          opf = self',
          __mixfix2(__mixfix1(filename, empty, READ_WRITE),
            empty,
            READ_WRITE,
            1)
            = self'),
       open(__mixfix1(filename, empty, mode), mode) = self')
      & if(0 = mode,
           if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
                & (__sel7_name(f) = filename),
              opf = self1',
              __mixfix2(__mixfix1(filename, empty, READ_WRITE),
                empty,
                READ_WRITE,
                1)
                = self1'),
           open(__mixfix1(filename, empty, mode), mode) = self1'))
     => (self' = self1')
    == true
Proof suspended

LP35  resume by cases
Please enter terms defining cases, terminated with a '  ' line, or '?' for
help
(0.String=mode)=false
..


Conjecture StringTheorem 8  Subgoals for proof by cases
New constant: modec
Case hypotheses:
  StringTheoremCaseHyp 5.1: (0 = modec) = false == true
  StringTheoremCaseHyp 5.2: not((0 = modec) = false) == true
Subgoal for cases
  StringTheorem.8.1:2:
    (if(0 = modec,
        if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
             & (__sel7_name(f) = filename),
           opf = self',
           __mixfix2(__mixfix1(filename, empty, READ_WRITE),
             empty,
             READ_WRITE,
             1)
             = self'),
        open(__mixfix1(filename, empty, modec), modec) = self')
       & if(0 = modec,
            if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
                 & (__sel7_name(f) = filename),
               opf = self1',
               __mixfix2(__mixfix1(filename, empty, READ_WRITE),
                 empty,
                 READ_WRITE,
                 1)
                 = self1'),
            open(__mixfix1(filename, empty, modec), modec) = self1'))
      => (self' = self1')
```

```
      == true

The current conjecture is subgoal StringTheorem.8.1.

Added hypothesis StringTheoremCaseHyp.5.1 to the system.

Deduction rule lp_not_is_true has been applied to equation
StringTheoremCaseHyp.5.1 to yield equation StringTheoremCaseHyp.5 1 1,
  0 = modec == false,
which implies StringTheoremCaseHyp.5.1.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.1:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
         opf = self',
         __mixfix2(__mixfix1(filename, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
     & if(0 = modec,
          if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
               & (__sel7_name(f) = filename),
             opf = self1',
             __mixfix2(__mixfix1(filename, empty, READ_WRITE),
               empty,
               READ_WRITE,
               1)
               = self1'),
          open(__mixfix1(filename, empty, modec), modec) = self1'))
    => (self' = self1')
    == true
  Current subgoal:
    ((open(__mixfix1(filename, empty, modec), modec) = self')
       & (open(__mixfix1(filename, empty, modec), modec) = self1'))
      => (self' = self1')
      == true
Proof suspended

LP36: resume by =>

Subgoal StringTheorem 8.1: Subgoal for proof of =>
New constants: filenamec, self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp.2:
     (open(__mixfix1(filenamec, empty, modec), modec) = self'c)
       & (open(__mixfix1(filenamec, empty, modec), modec) = self1'c)
       == true
Subgoal:
  StringTheorem.8.1.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.8.1.1.

Added hypothesis StringTheoremImpliesHyp.2 to the system.
```

```
Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.2 to yield the following equations, which imply
StringTheoremImpliesHyp 2.
  StringTheoremImpliesHyp.2.1:
     open(__mixfix1(filenamec, empty, modec), modec) = self'c == true
  StringTheoremImpliesHyp.2.2:
     open(__mixfix1(filenamec, empty, modec), modec) = self1'c == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.2.1 to yield equation StringTheoremImpliesHyp 2.1.1,
  open(__mixfix1(filenamec, empty, modec), modec) == self'c,
which implies StringTheoremImpliesHyp 2.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.2.2 to yield equation StringTheoremImpliesHyp.2 2.1,
  open(__mixfix1(filenamec, empty, modec), modec) == self1'c,
which implies StringTheoremImpliesHyp 2.2

Subgoal StringTheorem.8.1.1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is subgoal StringTheorem 8 1.

Subgoal StringTheorem 8 1:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
         opf = self',
         __mixfix2(__mixfix1(filename, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
    & if(0 = modec,
         if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
              & (__sel7_name(f) = filename),
            opf = self1',
            __mixfix2(__mixfix1(filename, empty, READ_WRITE),
              empty,
              READ_WRITE,
              1)
              = self1'),
         open(__mixfix1(filename, empty, modec), modec) = self1'))
   => (self' = self1')
   == true
[] Proved =>.

The current conjecture is subgoal StringTheorem.8.2.

Added hypothesis StringTheoremCaseHyp.5 2 to the system.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.5.2 to yield equation StringTheoremCaseHyp.5.2.1,
  0 == modec, which implies StringTheoremCaseHyp.5.2.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.2:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
```

```
          & (__sel7_name(f) = filename),,
        opf = self',
        __mixfix2(__mixfix1(filename, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
     & if(0 = modec,
          if((__mixfix2(f  __sel5_data(f), READ_WRITE, 1) = opf)
               & (__sel7_name(f) = filename),
             opf = self1',
             __mixfix2(__mixfix1(filename, empty, READ_WRITE),
               empty,
               READ_WRITE,
               1)
               = self1'),
          open(__mixfix1(filename, empty, modec), modec) = self1'))
    => (self' = self1')
    == true
  Current subgoal.
    (if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
          & (__sel7_name(f) = filename),
        opf = self',
        __mixfix2(__mixfix1(filename, empty, READ_WRITE), empty, READ_WRITE, 1)
          = self')
       & if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
             & (__sel7_name(f) = filename),
           opf = self1',
           __mixfix2(__mixfix1(filename, empty, READ_WRITE),
             empty,
             READ_WRITE,
             1)
             = self1'))
      => (self' = self1')
      == true
Proof suspended.

LP37  resume by cases
Please enter terms defining cases, terminated with a '..' line, or '?' for
help
(__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
         & (__sel7_name(f) = filename)
..


Subgoal StringTheorem 8.2: Subgoals for proof by cases
New constants: fc, opfc, filenamec
Case hypotheses:
  StringTheoremCaseHyp.6.1:
     (__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
       & (__sel7_name(fc) = filenamec)
       == true
  StringTheoremCaseHyp.6.2:
     not((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec))
       == true
Subgoal for cases:
  StringTheorem.8.2.1:2:
     (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
```

```
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
        empty,
        READ_WRITE,
        1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = selfi',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = selfi'))
  => (self' = selfi')
  == true


The current conjecture is subgoal StringTheorem 8 2 1.

Added hypothesis StringTheoremCaseHyp 6 1 to the system

Deduction rule lp_and_is_true has been applied to equation
StringTheoremCaseHyp 6.1 to yield the following equations, which imply
StringTheoremCaseHyp.6.1.
  StringTheoremCaseHyp.6.1.1:
     __mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc == true
  StringTheoremCaseHyp.6.1.2: __sel7_name(fc) = filenamec == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.6.1.1 to yield equation StringTheoremCaseHyp 6 1 1.1,
  __mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) == opfc,
which implies StringTheoremCaseHyp 6.1.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremCaseHyp.6.1.2 to yield equation StringTheoremCaseHyp.6 1.2 1,
  __sel7_name(fc) == filenamec,
which implies StringTheoremCaseHyp 6.1.2.

The equations cannot be ordered using the current ordering

Subgoal StringTheorem.8.2 1:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
        & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = selfi',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = selfi'))
   => (self' = selfi')
   == true
  Current subgoal:
    ((opfc = self') & (opfc = selfi')) => (self' = selfi') == true
Proof suspended.
```

```
LP38: resume by =>

Subgoal StringTheorem.8.2.1: Subgoal for proof of =>
New constants: self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp.3: (opfc = self'c) & (opfc = self1'c) == true
Subgoal:
  StringTheorem 8.2 1.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.8.2.1.1.

Added hypothesis StringTheoremImpliesHyp.3 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.3 to yield the following equations, which imply
StringTheoremImpliesHyp.3.
  StringTheoremImpliesHyp.3.1: opfc = self'c == true
  StringTheoremImpliesHyp.3.2: opfc = self1'c == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp 3.1 to yield equation StringTheoremImpliesHyp 3.1 1,
  opfc == self'c,
which implies StringTheoremImpliesHyp.3.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.3.2 to yield equation StringTheoremImpliesHyp.3.2.1,
  opfc == self1'c,
which implies StringTheoremImpliesHyp 3 2

Subgoal StringTheorem.8.2.1.1: self'c = self1'c == true
[] Proved by normalization

The current conjecture is subgoal StringTheorem.8 2.1

Subgoal StringTheorem.8.2.1:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
        & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
   => (self' = self1')
   == true
[] Proved =>.

The current conjecture is subgoal StringTheorem.8.2.2.

Added hypothesis StringTheoremCaseHyp.6.2 to the system.

Deduction rule lp_not_is_true has been applied to equation
StringTheoremCaseHyp.6.2 to yield equation StringTheoremCaseHyp.6.2.1,
  (__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
    & (__sel7_name(fc) = filenamec)
```

```
   == false,
which implies StringTheoremCaseHyp 6.2.

The equations cannot be ordered using the current ordering.

Subgoal StringTheorem.8.2.2:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
       & (__sel7_name(fc) = filenamec),
     opfc = self',
     __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self')
    & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
          & (__sel7_name(fc) = filenamec),
        opfc = self1',
        __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
   => (self' = self1')
   == true
  Current subgoal:
    ((__mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
      & (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'))
     => (self' = self1')
     == true
Proof suspended
LP40. resume by =>

Subgoal StringTheorem.8.2 2: Subgoal for proof of =>
New constants: self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp.4:
    (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self'c)
      & (__mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
          empty,
          READ_WRITE,
          1)
          = self1'c)
      == true
Subgoal:
  StringTheorem.8.2.2.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.8.2.2.1.

Added hypothesis StringTheoremImpliesHyp.4 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.4 to yield the following equations, which imply
StringTheoremImpliesHyp.4.
  StringTheoremImpliesHyp.4.1:
    __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
      = self'c
      == true
```

```
  StringTheoremImpliesHyp.4.2:
     __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
       = self1'c
       == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.4.1 to yield equation StringTheoremImpliesHyp.4.1.1,
  __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
    == self'c,
which implies StringTheoremImpliesHyp.4.1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp.4.2 to yield equation StringTheoremImpliesHyp.4.2.1,
  __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
    == self1'c,
which implies StringTheoremImpliesHyp.4.2.

Subgoal StringTheorem 8 2 2 1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is subgoal StringTheorem.8.2 2.

Subgoal StringTheorem 8.2.2:
  (if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
        & (__sel7_name(fc) = filenamec),
      opfc = self',
      __mixfix2(__mixfix1(filenamec, empty, READ_WRITE), empty, READ_WRITE, 1)
        = self')
     & if((__mixfix2(fc, __sel5_data(fc), READ_WRITE, 1) = opfc)
           & (__sel7_name(fc) = filenamec),
         opfc = self1',
         __mixfix2(__mixfix1(filenamec, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self1'))
    => (self' = self1')
    == true
[] Proved =>.

The current conjecture is subgoal StringTheorem.8.2.

Subgoal StringTheorem.8.2:
  (if(0 = modec,
      if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
           & (__sel7_name(f) = filename),
         opf = self',
         __mixfix2(__mixfix1(filename, empty, READ_WRITE),
           empty,
           READ_WRITE,
           1)
           = self'),
      open(__mixfix1(filename, empty, modec), modec) = self')
     & if(0 = modec,
          if((__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
               & (__sel7_name(f) = filename),
            opf = self1',
            __mixfix2(__mixfix1(filename, empty, READ_WRITE),
              empty,
              READ_WRITE,
              1)
```

```
            = self1'),
        open(__mixfix1(filename, empty, modec), modec) = self1'))
   => (self' = self1')
   == true
[] Proved by cases
   (__mixfix2(f, __sel5_data(f), READ_WRITE, 1) = opf)
    & (__sel7_name(f) = filename).

The current conjecture is StringTheorem.8.

Conjecture StringTheorem.8:
  (if(0 = mode,
      if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
         opf = self',
         open(create(filename, READ_WRITE), READ_WRITE) = self'),
      open(create(filename, mode), mode) = self')
    & if(0 = mode,
         if((open(f, READ_WRITE) = opf) & (__sel7_name(f) = filename),
            opf = self1',
            open(create(filename, READ_WRITE), READ_WRITE) = self1'),
         open(create(filename, mode), mode) = self1'))
   => (self' = self1')
   == true
[] Proved by cases (0 = mode) = false
```

Figure 27: LP proofs for completeness of RWFile.lcc constructor

```
Larch Prover (24 January 1994) logging on 20 November 1994 16.23:54 to
'/mnt/kbs1/BBRS/uman/thesis/lp/file/CompFileDest lplcg'.

LP2  prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help
  flush(self') = flush'(self')
  ..


The current conjecture is StringTheorem.3

Conjecture StringTheorem 3: flush(self') == flush'(self')
[] Proved by normalization.

Deleted equation StringTheorem.3, which reduced to an identity.

The equations cannot be ordered using the current ordering.

LP3: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (self' = flush(self) & self1' = flush(self)) => self' = self1'
  ..


The current conjecture is StringTheorem.4.

The equations cannot be ordered using the current ordering.
```

```
Conjecture StringTheorem.4:
  ((flush(self) = self') & (flush(self) = self1')) => (self' = self1') == true
  Current subgoal:
    ((__mixfix2(
        __mixfix1(__sel4_name(__sel3_file(self)),
          __sel6_data(self),
          __sel7_mode(__sel3_file(self))),
        __sel6_data(self),
        __sel8_mode(self),
        __sel9_fpointer(self))
        = self')
       & (__mixfix2(
            __mixfix1(__sel4_name(__sel3_file(self)),
              __sel6_data(self),
              __sel7_mode(__sel3_file(self))),
            __sel6_data(self),
            __sel8_mode(self),
            __sel9_fpointer(self))
            = self1'))
      => (self' = self1')
      == true
Proof suspended.

LP4: resume by =>

Conjecture StringTheorem 4  Subgoal for proof of =>
New constants. selfc, self'c, self1'c
Hypothesis:
  StringTheoremImpliesHyp 1:
     (__mixfix2(
        __mixfix1(__sel4_name(__sel3_file(selfc)),
          __sel6_data(selfc),
          __sel7_mode(__sel3_file(selfc))),
        __sel6_data(selfc),
        __sel8_mode(selfc),
        __sel9_fpointer(selfc))
        = self'c)
       & (__mixfix2(
            __mixfix1(__sel4_name(__sel3_file(selfc)),
              __sel6_data(selfc),
              __sel7_mode(__sel3_file(selfc))),
            __sel6_data(selfc),
            __sel8_mode(selfc),
            __sel9_fpointer(selfc))
            = self1'c)
       == true
Subgoal:
  StringTheorem.4.1: self'c = self1'c == true


The current conjecture is subgoal StringTheorem.4.1.

Added hypothesis StringTheoremImpliesHyp.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
StringTheoremImpliesHyp.1 to yield the following equations, which imply
StringTheoremImpliesHyp.1.
  StringTheoremImpliesHyp.1.1:
     __mixfix2(
       __mixfix1(__sel4_name(__sel3_file(selfc)),
         __sel6_data(selfc),
```

```
        __sel7_mode(__sel3_file(selfc))),
      __sel6_data(selfc),
      __sel8_mode(selfc),
      __sel9_fpointer(selfc))
      = self'c
      == true
  StringTheoremImpliesHyp 1.2:
    __mixfix2(
      __mixfix1(__sel4_name(__sel3_file(selfc)),
        __sel6_data(selfc),
        __sel7_mode(__sel3_file(selfc))),
      __sel6_data(selfc),
      __sel8_mode(selfc),
      __sel9_fpointer(selfc))
      = self1'c
      == true

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp 1.1 to yield equation StringTheoremImpliesHyp 1 1 1,
  __mixfix2(
    __mixfix1(__sel4_name(__sel3_file(selfc)),
      __sel6_data(selfc),
      __sel7_mode(__sel3_file(selfc))),
    __sel6_data(selfc),
    __sel8_mode(selfc),
    __sel9_fpointer(selfc))
    == self'c,
which implies StringTheoremImpliesHyp 1 1.

Deduction rule lp_equals_is_true has been applied to equation
StringTheoremImpliesHyp 1 2 to yield equation StringTheoremImpliesHyp 1.2 1,
  __mixfix2(
    __mixfix1(__sel4_name(__sel3_file(selfc)),
      __sel6_data(selfc),
      __sel7_mode(__sel3_file(selfc))),
    __sel6_data(selfc),
    __sel8_mode(selfc),
    __sel9_fpointer(selfc))
    == self1'c,
which implies StringTheoremImpliesHyp.1.2.

Subgoal StringTheorem 4 1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is StringTheorem.4.

Conjecture StringTheorem.4:
  ((flush(self) = self') & (flush(self) = self1')) => (self' = self1') == true
[] Proved =>.

The equations cannot be ordered using the current ordering.

LP5:

LP6: q
```

Figure 28: LP proofs for completeness of RWFile.lcc destructor

```
Larch Prover (24 January 1994) logging on 20 November 1994 12:26:44 to
'/mnt/kbsi/BBRS/uman/thesis/lp/file/CompFileRead1.lplog'.

LP2: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
read'(self, i, j)=read(self,i, j)
..


The current conjecture is CompFileRead1Theorem.1.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead1Theorem.1: read'(self, i, j) == read(self, i, j)
Proof suspended.

LP3: display deduction Deque
LP5: display deduction File

Deduction rules:

File.2   when __sel11_name(f1) == __sel11_name(f2),
              __sel3_data(f1) == __sel3_data(f2),
              __sel8_mode(f1) == __sel8_mode(f2)
         yield f1 == f2
File.10  when __sel7_file(r1) == __sel7_file(r2),
              __sel4_data(r1) == __sel4_data(r2),
              __sel9_mode(r1) == __sel9_mode(r2),
              __sel10_fpointer(r1) == __sel10_fpointer(r2)
         yield r1 == r2
File.30: when __sel6_ofile(r3) == __sel6_ofile(r4),
              __sel5_reddata(r3) == __sel5_reddata(r4)
         yield r3 == r4


LP6: instantiate r3 by read(opf, i, j), r4 by read'(opf, i, j) in File.30

Deduction rule File.30 has been instantiated to deduction rule File.30.1,
  when __sel6_ofile(read(opf, i, j)) == __sel6_ofile(read'(opf, i, j)),
       __sel5_reddata(read(opf, i, j)) == __sel5_reddata(read'(opf, i, j))
  yield read(opf, i, j) == read'(opf, i, j)

Deduction rule File.30.1 was normalized to equation File.30.1.1,
  read(opf, i, j) == read'(opf, i, j)

Conjecture CompFileRead1Theorem.1: read'(self, i, j) == read(self, i, j)
[] Proved by normalization.
LP7: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  toByte'(t) = toByte(t)
  ..


The current conjecture is CompFileRead1Theorem.2.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead1Theorem.2: toByte'(t) == toByte(t)
Proof suspended.
```

```
LP9: instantiate d1 by toByte'(t), d2 by toByte(t) in Types.1

Deduction rule Types.1 has been instantiated to deduction rule Types.1.1,
  when toType(toByte'(t)) == toType(toByte(t)) yield toByte'(t) == toByte(t)

Deduction rule Types.1.1 was normalized to equation Types.1.1.1,
  toByte'(t) == toByte(t)

Conjecture CompFileRead1Theorem.2: toByte'(t) == toByte(t)
[] Proved by normalization.

LP10: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  len'(q) = len(q)
  ..


The current conjecture is CompFileRead1Theorem.3.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead1Theorem 3: len'(q) == len(q)
Proof suspended.

LP19: resume by induction on q using Deque 1

Conjecture CompFileRead1Theorem 3: Subgoals for proof by induction on 'q'
Basis subgoal:
  CompFileRead1Theorem.3.1: len'(empty) == len(empty)
Induction constant: qc
Induction hypothesis:
  CompFileRead1TheoremInductHyp.1: len'(qc) == len(qc)
Induction subgoal:
  CompFileRead1Theorem.3.2: len'(qc \postcat b) == len(qc \postcat b)


The current conjecture is subgoal CompFileRead1Theorem.3.1.

Subgoal CompFileRead1Theorem 3.1: len'(empty) == len(empty)
[] Proved by normalization.

The current conjecture is subgoal CompFileRead1Theorem 3 2.

Added hypothesis CompFileRead1TheoremInductHyp.1 to the system.

Subgoal CompFileRead1Theorem.3.2: len'(qc \postcat b) == len(qc \postcat b)
[] Proved by normalization.

The current conjecture is CompFileRead1Theorem.3.

Conjecture CompFileRead1Theorem.3: len'(q) == len(q)
[] Proved by induction on 'q'.
LP21: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (toByte(t') = __sel5_reddata(read(self, len(toByte(t)),
    __sel10_fpointer(self))) & __sel10_fpointer(self') =
    (__sel10_fpointer(self) + len(toByte(t))) & (toByte(t1') =
    __sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self))) &
    __sel10_fpointer(self') = (__sel10_fpointer(self) + len(toByte(t))))) =>
```

```
      t' = t1'
    ..

The current conjecture is CompFileRead1Theorem.4.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead1Theorem.4.
  ((__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
      = toByte(t'))
     & (__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
         = toByte(t1'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self'))
     & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
    => (t' = t1')
    == true
  Current subgoal:
    ((prefix(removePrefix(__sel4_data(self), __sel10_fpointer(self)),
        len(toByte(t)))
       = toByte(t'))
      & (prefix(removePrefix(__sel4_data(self), __sel10_fpointer(self)),
           len(toByte(t)))
           = toByte(t1'))
      & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
     => (t' = t1')
     == true
Proof suspended.

LP22.  resume by =>

Conjecture CompFileRead1Theorem.4: Subgoal for proof of =>
New constants  selfc, tc, t'c, t1'c, self'c
Hypothesis
  CompFileRead1TheoremImpliesHyp.1:
    (prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
       len(toByte(tc)))
       = toByte(t'c))
      & (prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
           len(toByte(tc)))
           = toByte(t1'c))
      & ((__sel10_fpointer(selfc) + len(toByte(tc)))
           = __sel10_fpointer(self'c))
      == true
Subgoal.
  CompFileRead1Theorem.4.1: t'c = t1'c == true


The current conjecture is subgoal CompFileRead1Theorem.4.1.

Added hypothesis CompFileRead1TheoremImpliesHyp.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1 to yield the following equations, which imply
CompFileRead1TheoremImpliesHyp.1.
  CompFileRead1TheoremImpliesHyp.1.1:
    prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
      len(toByte(tc)))
      = toByte(t'c)
      == true
  CompFileRead1TheoremImpliesHyp.1.2:
```

```
      prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
        len(toByte(tc)))
        = toByte(t1'c)
        == true
  CompFileRead1TheoremImpliesHyp.1.3.
      (__sel10_fpointer(selfc) + len(toByte(tc))) = __sel10_fpointer(self'c)
        == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1.1 to yield equation
CompFileRead1TheoremImpliesHyp.1.1.1,
  prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
    len(toByte(tc)))
    == toByte(t'c),
which implies CompFileRead1TheoremImpliesHyp.1.1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2 to yield equation
CompFileRead1TheoremImpliesHyp.1.2 1,
  prefix(removePrefix(__sel4_data(selfc), __sel10_fpointer(selfc)),
    len(toByte(tc)))
    == toByte(t1'c),
which implies CompFileRead1TheoremImpliesHyp 1.2.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead1TheoremImpliesHyp.1.3 to yield equation
CompFileRead1TheoremImpliesHyp 1.3.1,
  __sel10_fpointer(selfc) + len(toByte(tc)) == __sel10_fpointer(self'c),
which implies CompFileRead1TheoremImpliesHyp.1.3.

Deduction rule Types.2 has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2 1 to yield equation
CompFileRead1TheoremImpliesHyp 1 2.1.1, t'c == t1'c,
which implies CompFileRead1TheoremImpliesHyp.1.2.1

Deduction rule CompFileRead1.1 has been applied to equation
CompFileRead1TheoremImpliesHyp.1.2.1 to yield equation
CompFileRead1TheoremImpliesHyp.1.2.1.2, t'c == t1'c,
which implies CompFileRead1TheoremImpliesHyp.1.2.1.

Subgoal CompFileRead1Theorem 4.1: t'c = t1'c == true
[] Proved by normalization.

The current conjecture is CompFileRead1Theorem.4.

Conjecture CompFileRead1Theorem.4:
  ((__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
      = toByte(t'))
    & (__sel5_reddata(read(self, len(toByte(t)), __sel10_fpointer(self)))
        = toByte(t1'))
    & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self'))
    & ((__sel10_fpointer(self) + len(toByte(t))) = __sel10_fpointer(self')))
   => (t' = t1')
   == true
[] Proved =>.
```

Figure 29: LP proofs for completeness of RWFile.lcc Read(&char) member-function

```
Larch Prover (24 January 1994) logging on 20 November 1994 17:54:31 to
'/mnt/kbs1/BBRS/uman/thesis/lp/file/CompFileRead2.lplog'.

LP0.1.8: declare variables
  t': character
  t1': character
  self'. RWFile
  self1. RWFile
  self1': RWFile
  j. Int
  ind: Int
  i1: String
  i': String
  i1': String
  s1: Data
  s2  Data
  d: Data
  q  Data
  e: Byte
  e1. Byte
  t: character
  t1. character
  dat· Data
  self. RWFile
  opf· RWFile
  p  Int
  r. String
  count: Int
  .


LP5· prove
Please enter a conjecture to prove, terminated with a ' .' line, or '?' for
help·
read'(self, j, p:Int)=read(self, j, p:Int)
..

LP14· display File.30

Deduction rules

File 30· when __sel6_ofile(r3) == __sel6_ofile(r4),
             __sel5_reddata(r3) == __sel5_reddata(r4)
         yield r3 == r4

LP16: instantiate r3 by read'(self, j, p:Int), r4 by read(self, j, p:Int) in File.30

Deduction rule File.30 has been instantiated to deduction rule File.30.2,
  when __sel6_ofile(read'(self, j, p)) == __sel6_ofile(read(self, j, p)),
       __sel5_reddata(read'(self, j, p)) == __sel5_reddata(read(self, j, p))
  yield read'(self, j, p) == read(self, j, p)

Deduction rule File.30.2 was normalized to equation File.30.2.1,
  read'(self, j, p) == read(self, j, p)

Conjecture CompFileRead2Theorem.1: read'(self, j, p) == read(self, j, p)
[] Proved by normalization.

Deleted equation CompFileRead2.3, which reduced to an identity.

Deleted equation CompFileRead2.4, which reduced to an identity.
```

```
Deleted equation CompFileRead2.12, which reduced to an identity.

The equations cannot be ordered using the current ordering.

LP17: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  toByte'(t) = toByte(t)
  ..


The current conjecture is lemma CompFileRead2Theorem.4.

The equations cannot be ordered using the current ordering.

Lemma CompFileRead2Theorem.4: toByte'(t) == toByte(t)
Proof suspended.

LP24: display Types.1

Types.1:     when toType(d1) == toType(d2) yield d1 == d2

LP25: instantiate d1 by toByte'(t), d2 by toByte(t) in Types.1

Deduction rule Types 1 has been instantiated to deduction rule Types 1 1,
  when toType(toByte'(t)) == toType(toByte(t)) yield toByte'(t) == toByte(t)

Deduction rule Types.1.1 was normalized to equation Types.1 1 1,
  toByte'(t) == toByte(t)

Lemma CompFileRead2Theorem.4: toByte'(t) == toByte(t)
[] Proved by normalization

LP8: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (((if(ind >= 0:Int & ind <= count, toByte(__prefix2(ptr + ind)) =
    __sel5_reddata(read(self, len(toByte(__prefix2(ptr))),
    __sel10_fpointer(self) + (ind *
len(toByte(__prefix2(ptr))))))), __prefix2(ptr
    + ind) = __prefix2(ptr + ind)) & __sel10_fpointer(self') =
    (__sel10_fpointer(self) + count))) & ((if(ind >= 0:Int
 & ind <= count,
    toByte(__prefix2(ptr + ind)) = __sel5_reddata(read(self,
    len(toByte(__prefix2(ptr))), __sel10_fpointer(self) + (ind *
    len(toByte(__prefix2(ptr)))))), __prefix2(ptr + ind) = __prefix2(ptr + ind)) &
    __sel10_fpointer(self1') = (__sel10_fpointer(self) + count)))) =>
    __sel10_fpointer(self') = __sel10_fpointer(self1')
  ..


The current conjecture is CompFileRead2Theorem.2.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead2Theorem.2:
  (((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
    & ((__sel10_fpointer(self) + count) = __sel10_fpointer(self1'))
    & if((ind <= count) & (ind >= 0),
        __sel5_reddata(
```

```
              read(self,
                   len(toByte(__prefix2(ptr))),
                   (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
              = toByte(__prefix2(ptr + ind)),
            __prefix2(ptr + ind) = __prefix2(ptr + ind))
       & if((ind <= count) & (ind >= 0),
            __sel5_reddata(
              read(self,
                   len(toByte(__prefix2(ptr))),
                   (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
              = toByte(__prefix2(ptr + ind)),
            __prefix2(ptr + ind) = __prefix2(ptr + ind)))
      => (__sel10_fpointer(self') = __sel10_fpointer(self1'))
      == true
  Current subgoal:
    (((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
       & ((__sel10_fpointer(self) + count) = __sel10_fpointer(self1'))
       & ((((0 < ind) | (0 = ind)) & ((ind < count) | (count = ind)))
            => (prefix(
                  removePrefix(__sel4_data(self),
                    (len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr))))
                       * ind)
                      + __sel10_fpointer(self)),
                  len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                  = toByte(
                      __mixfix6(
                        __sel2_locs(set_idx(ptr, __sel1_idx(ptr) + ind)),
                        __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + ind)))))))
      => (__sel10_fpointer(self') = __sel10_fpointer(self1'))
      == true
Proof suspended

LP10  resume by =>

Conjecture CompFileRead2Theorem 2  Subgoal for proof of =>
New constants  selfc, countc, self'c, self1'c, indc, ptrc
Hypothesis
  CompFileRead2TheoremImpliesHyp 1:
     ((__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self'c))
       & ((__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self1'c))
       & ((((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
            => (prefix(
                  removePrefix(__sel4_data(selfc),
                    (indc
                       * len(toByte(
                               __mixfix6(__sel2_locs(ptrc),
                                 __sel1_idx(ptrc)))))
                      + __sel10_fpointer(selfc)),
                  len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
                  = toByte(
                      __mixfix6(
                        __sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
                        __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc))))))
       == true
Subgoal:
  CompFileRead2Theorem 2.1:
     __sel10_fpointer(self'c) = __sel10_fpointer(self1'c) == true


The current conjecture is subgoal CompFileRead2Theorem.2.1.
```

```
Added hypothesis CompFileRead2TheoremImpliesHyp.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.1 to yield the following equations, which imply
CompFileRead2TheoremImpliesHyp.1.
  CompFileRead2TheoremImpliesHyp.1.1:
     (__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self'c) == true
  CompFileRead2TheoremImpliesHyp.1.2:
     (__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self1'c) == true
  CompFileRead2TheoremImpliesHyp.1.3:
     (((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
       => (prefix(
            removePrefix(__sel4_data(selfc),
              (indc
                 * len(toByte(
                         __mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
                + __sel10_fpointer(selfc)),
            len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
            = toByte(
                __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
                  __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc)))))
       == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.1.1 to yield equation
CompFileRead2TheoremImpliesHyp.1 1.1,
  __sel10_fpointer(selfc) + countc == __sel10_fpointer(self'c),
which implies CompFileRead2TheoremImpliesHyp.1 1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.1.2 to yield equation
CompFileRead2TheoremImpliesHyp.1.2.1,
  __sel10_fpointer(selfc) + countc == __sel10_fpointer(self1'c),
which implies CompFileRead2TheoremImpliesHyp 1.2.

Subgoal CompFileRead2Theorem 2.1.
  __sel10_fpointer(self'c) = __sel10_fpointer(self1'c) == true
[] Proved by normalization.

The current conjecture is CompFileRead2Theorem.2.

Conjecture CompFileRead2Theorem 2:
  (((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
     & ((__sel10_fpointer(self) + count) = __sel10_fpointer(self1'))
     & if((ind <= count) & (ind >= 0),
          __sel5_reddata(
            read(self,
                 len(toByte(__prefix2(ptr))),
                 (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
            = toByte(__prefix2(ptr + ind)),
          __prefix2(ptr + ind) = __prefix2(ptr + ind))
     & if((ind <= count) & (ind >= 0),
          __sel5_reddata(
            read(self,
                 len(toByte(__prefix2(ptr))),
                 (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
            = toByte(__prefix2(ptr + ind)),
          __prefix2(ptr + ind) = __prefix2(ptr + ind)))
    => (__sel10_fpointer(self') = __sel10_fpointer(self1'))
    == true
[] Proved =>.
```

```
LP14: declare variable
Please enter variable declarations, terminated with a '..' line, or '?' for
help:
ptr1:String
..

LP16: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (((if(ind >= 0:Int & ind <= count, toByte(__prefix2(ptr + ind)) =
    __sel5_reddata(read(self, len(toByte(__prefix2(ptr))),
    __sel10_fpointer(self) + (ind * len(toByte(__prefix2(ptr))))))), __prefix2(ptr
    + ind) = __prefix2(ptr + ind)) & __sel10_fpointer(self') =
    (__sel10_fpointer(self) + count))) & ((if(ind >= 0:Int & ind <= count,
    toByte(__prefix2(ptr1 + ind)) = __sel5_reddata(read(self,
    len(toByte(__prefix2(ptr))), __sel10_fpointer(self) + (ind *
    len(toByte(__prefix2(ptr)))))), __prefix2(ptr1 + ind) = __prefix2(ptr + ind)) &
    __sel10_fpointer(self') = (__sel10_fpointer(self) + count)))) =>
    __prefix2(ptr + ind) = __prefix2(ptr1 + ind)
  ..


The current conjecture is CompFileRead2Theorem.3.

The equations cannot be ordered using the current ordering.

Conjecture CompFileRead2Theorem.3:
  (((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
     & ((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
     & if((ind <= count) & (ind >= 0),
          __sel5_reddata(
            read(self,
                 len(toByte(__prefix2(ptr))),
                 (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
            = toByte(__prefix2(ptr1 + ind)),
          __prefix2(ptr1 + ind) = __prefix2(ptr + ind))
     & if((ind <= count) & (ind >= 0),
          __sel5_reddata(
            read(self,
                 len(toByte(__prefix2(ptr))),
                 (len(toByte(__prefix2(ptr))) * ind) + __sel10_fpointer(self)))
            = toByte(__prefix2(ptr + ind)),
          __prefix2(ptr + ind) = __prefix2(ptr + ind)))
    => (__prefix2(ptr1 + ind) = __prefix2(ptr + ind))
    == true
  Current subgoal:
    (((__sel10_fpointer(self) + count) = __sel10_fpointer(self'))
       & ((((0 < ind) | (0 = ind)) & ((ind < count) | (count = ind)))
            => (prefix(
                  removePrefix(__sel4_data(self),
                    (len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr))))
                       * ind)
                      + __sel10_fpointer(self)),
                  len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                  = toByte(
                      __mixfix6(
                        __sel2_locs(set_idx(ptr, __sel1_idx(ptr) + ind)),
```

```
                    __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + ind))))))
      & if(((0 < ind) | (0 = ind)) & ((ind < count) | (count = ind)),
           prefix(
             removePrefix(__sel4_data(self),
               (len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr))))
                  * ind)
                 + __sel10_fpointer(self)),
             len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
             = toByte(
                 __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + ind)),
                   __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + ind)))),
           __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + ind)),
             __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + ind)))
             = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + ind)),
                 __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + ind)))))
      => (__mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + ind)),
           __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + ind)))
           = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + ind)),
               __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + ind))))
      == true
Proof suspended.

LP24: resume by cases
Please enter terms defining cases, terminated with a '..' line, or '?' for
help:
((0:Int < ind) | (0.Int = ind)) & ((ind < count) | (count = ind))
..


Conjecture CompFileRead2Theorem.3: Subgoals for proof by cases
New constants: indc, countc
Case hypotheses:
  CompFileRead2TheoremCaseHyp.2.1:
     ((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)) == true
  CompFileRead2TheoremCaseHyp 2.2:
     not(((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
       == true
Subgoal for cases:
  CompFileRead2Theorem.3.1:2:
     (((__sel10_fpointer(self) + countc) = __sel10_fpointer(self'))
        & ((((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
             => (prefix(
                   removePrefix(__sel4_data(self),
                     (indc
                        * len(toByte(
                                __mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                       + __sel10_fpointer(self)),
                   len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                   = toByte(
                       __mixfix6(
                         __sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                         __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))))
        & if(((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)),
             prefix(
               removePrefix(__sel4_data(self),
                 (indc
                    * len(toByte(
                            __mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                   + __sel10_fpointer(self)),
               len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
               = toByte(
```

```
                __mixfix6(
                  __sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
                  __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))),
              __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
                __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
                = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                    __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc)))))
        => (__mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
              __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
              = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                  __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))
        == true


The current conjecture is subgoal CompFileRead2Theorem.3.1.

Added hypothesis CompFileRead2TheoremCaseHyp.2.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
CompFileRead2TheoremCaseHyp 2.1 to yield the following equations, which imply
CompFileRead2TheoremCaseHyp 2 1
  CompFileRead2TheoremCaseHyp.2.1 1  (0 < indc) | (0 = indc) == true
  CompFileRead2TheoremCaseHyp.2 1.2: (indc < countc) | (countc = indc) == true

The equations cannot be ordered using the current ordering.

Subgoal CompFileRead2Theorem.3.1
  (((__sel10_fpointer(self) + countc) = __sel10_fpointer(self'))
     & ((((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
          => (prefix(
                removePrefix(__sel4_data(self),
                  (indc
                     * len(toByte(
                             __mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                    + __sel10_fpointer(self)),
                len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr))))
                = toByte(
                    __mixfix6(
                      __sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                      __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))))
     & if(((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)),
          prefix(
            removePrefix(__sel4_data(self),
              (indc
                 * len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                + __sel10_fpointer(self)),
            len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
            = toByte(
                __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
                  __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))),
          __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
            __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
            = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc)))))
    => (__mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
          __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
          = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
              __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))
    == true
  Current subgoal:
    ((prefix(
```

```
       removePrefix(__sel4_data(self),
         (indc * len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
           + __sel10_fpointer(self)),
       len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
       = toByte(
           __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
             __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))))
      & (prefix(
           removePrefix(__sel4_data(self),
             (indc
                * len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
               + __sel10_fpointer(self)),
           len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
           = toByte(
               __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                 __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc)))))
      & ((__sel10_fpointer(self) + countc) = __sel10_fpointer(self')))
     => (__mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
           __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
           = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
               __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))
     == true
Proof suspended.

LP25: resume by =>

Subgoal CompFileRead2Theorem 3.1: Subgoal for proof of =>
New constants: selfc, ptrc, ptr1c, self'c
Hypothesis:
  CompFileRead2TheoremImpliesHyp.2:
     (prefix(
        removePrefix(__sel4_data(selfc),
          (indc * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
            + __sel10_fpointer(selfc)),
        len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
        = toByte(
            __mixfix6(__sel2_locs(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)),
              __sel1_idx(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)))))
       & (prefix(
            removePrefix(__sel4_data(selfc),
              (indc
                 * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
                + __sel10_fpointer(selfc)),
            len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
            = toByte(
                __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
                  __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc)))))
       & ((__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self'c))
       == true
Subgoal:
  CompFileRead2Theorem.3.1.1:
     __mixfix6(__sel2_locs(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)),
       __sel1_idx(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)))
       = __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
           __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc)))
       == true


The current conjecture is subgoal CompFileRead2Theorem.3.1.1.

Added hypothesis CompFileRead2TheoremImpliesHyp.2 to the system.
```

```
Deduction rule lp_and_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.2 to yield the following equations, which imply
CompFileRead2TheoremImpliesHyp.2.
  CompFileRead2TheoremImpliesHyp.2.1:
     prefix(
       removePrefix(__sel4_data(selfc),
         (indc * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
           + __sel10_fpointer(selfc)),
       len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
       = toByte(
           __mixfix6(__sel2_locs(set_idx(ptric, __sel1_idx(ptric) + indc)),
             __sel1_idx(set_idx(ptric, __sel1_idx(ptric) + indc))))
       == true
  CompFileRead2TheoremImpliesHyp.2.2:
     prefix(
       removePrefix(__sel4_data(selfc),
         (indc * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
           + __sel10_fpointer(selfc)),
       len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
       = toByte(
           __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
             __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc))))
       == true
  CompFileRead2TheoremImpliesHyp.2.3:
     (__sel10_fpointer(selfc) + countc) = __sel10_fpointer(self'c) == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.2.1 to yield equation
CompFileRead2TheoremImpliesHyp.2.1.1,
  prefix(
    removePrefix(__sel4_data(selfc),
      (indc * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
        + __sel10_fpointer(selfc)),
    len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
    ==
    toByte(
      __mixfix6(__sel2_locs(set_idx(ptric, __sel1_idx(ptric) + indc)),
        __sel1_idx(set_idx(ptric, __sel1_idx(ptric) + indc)))),
which implies CompFileRead2TheoremImpliesHyp.2.1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.2.2 to yield equation
CompFileRead2TheoremImpliesHyp.2.2.1,
  prefix(
    removePrefix(__sel4_data(selfc),
      (indc * len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
        + __sel10_fpointer(selfc)),
    len(toByte(__mixfix6(__sel2_locs(ptrc), __sel1_idx(ptrc)))))
    ==
    toByte(
      __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
        __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc)))),
which implies CompFileRead2TheoremImpliesHyp.2.2.

Deduction rule lp_equals_is_true has been applied to equation
CompFileRead2TheoremImpliesHyp.2.3 to yield equation
CompFileRead2TheoremImpliesHyp.2.3.1,
  __sel10_fpointer(selfc) + countc == __sel10_fpointer(self'c),
which implies CompFileRead2TheoremImpliesHyp.2.3.
Subgoal CompFileRead2Theorem.3.1.1:
```

```
    __mixfix6(__sel2_locs(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)),
      __sel1_idx(set_idx(ptr1c, __sel1_idx(ptr1c) + indc)))
      = __mixfix6(__sel2_locs(set_idx(ptrc, __sel1_idx(ptrc) + indc)),
          __sel1_idx(set_idx(ptrc, __sel1_idx(ptrc) + indc)))
      == true

[] Proved by normalization

Subgoal CompFileRead2Theorem.3.1
  (((__sel10_fpointer(self) + countc) = __sel10_fpointer(self'))
    & ((((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)))
        => (prefix(
              removePrefix(__sel4_data(self),
                (indc
                   * len(toByte(
                           __mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
                  + __sel10_fpointer(self)),
              len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
              = toByte(
                  __mixfix6(
                    __sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
                    __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))))
    & if(((0 < indc) | (0 = indc)) & ((indc < countc) | (countc = indc)),
        prefix(
          removePrefix(__sel4_data(self),
            (indc
               * len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
              + __sel10_fpointer(self)),
          len(toByte(__mixfix6(__sel2_locs(ptr), __sel1_idx(ptr)))))
          = toByte(
              __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
                __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))),
        __mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
          __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
          = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
              __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc)))))
    => (__mixfix6(__sel2_locs(set_idx(ptr1, __sel1_idx(ptr1) + indc)),
        __sel1_idx(set_idx(ptr1, __sel1_idx(ptr1) + indc)))
        = __mixfix6(__sel2_locs(set_idx(ptr, __sel1_idx(ptr) + indc)),
            __sel1_idx(set_idx(ptr, __sel1_idx(ptr) + indc))))
    == true

[] Proved =>.
```

Figure 30: LP proofs for completeness of RWFile.lcc Read(Char *, size_t) member-function

```
Larch Prover (24 January 1994) logging on 20 November 1994 12:17:42 to .
'/mnt/kbs1/BBRS/uman/thesis/lp/file/CompFileWrite1.lplog'.


LP0.1.15: prove
  open'(f, m) == open(f, m)
  ..


The current conjecture is CompFileWrite1Theorem.1.

The equations cannot be ordered using the current ordering.

Conjecture CompFileWrite1Theorem.1: open'(f, m) == open(f, m)
Proof suspended.

LP3: resume by induction on m

There is more than one applicable induction rule.  You must pick one.

LP4: resume by induction on m using File.19

Conjecture CompFileWrite1Theorem 1. Subgoals for proof by induction on 'm'
Basis subgoals:
  CompFileWrite1Theorem.1.1: open'(f, 0) == open(f, 0)
  CompFileWrite1Theorem 1.2: open'(f, READ) == open(f, READ)
  CompFileWrite1Theorem.1.3: open'(f, WRITE) == open(f, WRITE)
  CompFileWrite1Theorem 1 4  open'(f, READ_WRITE) == open(f, READ_WRITE)
The induction step is vacuous.


The current conjecture is subgoal CompFileWrite1Theorem 1.1.

Subgoal CompFileWrite1Theorem.1.1: open'(f, 0) == open(f, 0)
[] Proved by normalization

The current conjecture is subgoal CompFileWrite1Theorem.1.2

Subgoal CompFileWrite1Theorem.1.2: open'(f, READ) == open(f, READ)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite1Theorem 1.3.

Subgoal CompFileWrite1Theorem.1.3: open'(f, WRITE) == open(f, WRITE)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite1Theorem.1.4.

Subgoal CompFileWrite1Theorem.1.4: open'(f, READ_WRITE) == open(f, READ_WRITE)
[] Proved by normalization.

The current conjecture is CompFileWrite1Theorem.1.

Conjecture CompFileWrite1Theorem.1: open'(f, m) == open(f, m)
[] Proved by induction on 'm'.

LP11: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
write'(opf, dt, count)=write(opf, dt, count)
..
```

```
The current conjecture is CompFileWrite1Theorem.3.

Conjecture CompFileWrite1Theorem.3:
  write'(opf, dt, count) == write(opf, dt, count)
[] Proved by normalization.

LP12: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  toByte(t) = toByte(t)
  ..


The current conjecture is CompFileWrite1Theorem.4.

Conjecture CompFileWrite1Theorem.4: toByte(t) == toByte(t)
[] Proved by normalization.

LP23: declare variable pt:String

LP25: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (self' = write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) & result &
    (self1' = write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) &
    result)) => self' = self1'
  ..


The current conjecture is CompFileWrite1Theorem 5

The equations cannot be ordered using the current ordering.

Conjecture CompFileWrite1Theorem.5:
  ((write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) = self')
     & (write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) = self1')
     & result
     & result)
    => (self' = self1')
    == true
  Current subgoal:
    ((__mixfix2(__sel7_file(self),
        (prefix(__sel6_data(self), __sel8_fpointer(self))
           || toByte(__mixfix6(__sel2_locs(pt), __sel1_idx(pt))))
          || removePrefix(__sel6_data(self),
               __sel8_fpointer(self)
                 + len(toByte(__mixfix6(__sel2_locs(pt), __sel1_idx(pt))))),
        __sel4_mode(self),
        __sel8_fpointer(self)
          + len(toByte(__mixfix6(__sel2_locs(pt), __sel1_idx(pt)))))
        = self')
       & (__mixfix2(__sel7_file(self),
            (prefix(__sel6_data(self), __sel8_fpointer(self))
               || toByte(__mixfix6(__sel2_locs(pt), __sel1_idx(pt))))
              || removePrefix(__sel6_data(self),
                   __sel8_fpointer(self)
                     + len(toByte(
                               __mixfix6(__sel2_locs(pt), __sel1_idx(pt))))),
            __sel4_mode(self),
```

```
          __sel8_fpointer(self)
            + len(toByte(__mixfix6(__sel2_locs(pt), __sel1_idx(pt)))))
          = self1')
       & result)
      => (self' = self1')
      == true
Proof suspended
LP27: resume by =>

Conjecture CompFileWrite1Theorem.5: Subgoal for proof of =>
New constants. selfc, ptc, self'c, self1'c, resultc
Hypothesis.
  CompFileWrite1TheoremImpliesHyp.1:
     (__mixfix2(__sel7_file(selfc),
       (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
          || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
        || removePrefix(__sel6_data(selfc),
             __sel8_fpointer(selfc)
               + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
       __sel4_mode(selfc),
       __sel8_fpointer(selfc)
         + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc)))))
       = self'c)
      & (__mixfix2(__sel7_file(selfc),
          (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
             || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
           || removePrefix(__sel6_data(selfc),
                __sel8_fpointer(selfc)
                  + len(toByte(
                          __mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
          __sel4_mode(selfc),
          __sel8_fpointer(selfc)
            + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc)))))
          = self1'c)
      & resultc
      == true
Subgoal:
  CompFileWrite1Theorem.5.1: self'c = self1'c == true


The current conjecture is subgoal CompFileWrite1Theorem.5.1.

Added hypothesis CompFileWrite1TheoremImpliesHyp.1 to the system

Deduction rule lp_and_is_true has been applied to equation
CompFileWrite1TheoremImpliesHyp.1 to yield the following equations, which imply
CompFileWrite1TheoremImpliesHyp.1.
  CompFileWrite1TheoremImpliesHyp.1.1:
     __mixfix2(__sel7_file(selfc),
       (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
          || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
        || removePrefix(__sel6_data(selfc),
             __sel8_fpointer(selfc)
               + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
       __sel4_mode(selfc),
       __sel8_fpointer(selfc)
         + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc)))))
       = self'c
       == true
  CompFileWrite1TheoremImpliesHyp.1.2:
     __mixfix2(__sel7_file(selfc),
```

```
    .  (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
          || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
        || removePrefix(__sel6_data(selfc),
             __sel8_fpointer(selfc)
               + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
       __sel4_mode(selfc),
       __sel8_fpointer(selfc)
         + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
       = self1'c
       == true
  CompFileWrite1TheoremImpliesHyp.1.3: resultc == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileWrite1TheoremImpliesHyp.1.1 to yield equation
CompFileWrite1TheoremImpliesHyp.1.1.1,
  __mixfix2(__sel7_file(selfc),
    (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
       || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
      || removePrefix(__sel6_data(selfc),
           __sel8_fpointer(selfc)
             + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
    __sel4_mode(selfc),
    __sel8_fpointer(selfc)
      + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
    == self'c,
which implies CompFileWrite1TheoremImpliesHyp 1.1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileWrite1TheoremImpliesHyp 1.2 to yield equation
CompFileWrite1TheoremImpliesHyp.1.2.1,
  __mixfix2(__sel7_file(selfc),
    (prefix(__sel6_data(selfc), __sel8_fpointer(selfc))
       || toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
      || removePrefix(__sel6_data(selfc),
           __sel8_fpointer(selfc)
             + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))),
    __sel4_mode(selfc),
    __sel8_fpointer(selfc)
      + len(toByte(__mixfix6(__sel2_locs(ptc), __sel1_idx(ptc))))
    == self1'c,
which implies CompFileWrite1TheoremImpliesHyp.1.2.

Subgoal CompFileWrite1Theorem.5.1: self'c = self1'c == true
[] Proved by normalization.

The current conjecture is CompFileWrite1Theorem.5.

Conjecture CompFileWrite1Theorem.5:
  ((write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) = self')
     & (write(self, toByte(__prefix2(pt)), __sel8_fpointer(self)) = self1')
     & result
     & result)
    => (self' = self1')
    == true
[] Proved =>.
```

Figure 31: LP proofs for completeness of RWFile.lcc Write(char) member-function

```
Larch Prover (24 January 1994) logging on 20 November 1994 11:56:59 to
'/mnt/kbs1/BBRS/uman/thesis/lp/file/CompFileWrite2.lplog'.

LP4: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
maxIndex'(ptr)=maxIndex(ptr)
..

The current conjecture is CompFileWrite2Theorem.1.

Conjecture CompFileWrite2Theorem.1: maxIndex'(ptr) == maxIndex(ptr)
[] Proved by normalization.

Deleted equation CompFileWrite2Theorem.1, which reduced to an identity.

The equations cannot be ordered using the current ordering.

LP5: prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  open'(f, m) == open(f, m)
 .


The current conjecture is CompFileWrite2Theorem 2.
Conjecture CompFileWrite2Theorem 2. open'(f, m) == open(f, m)
Proof suspended.

LP8: resume by induction on m using File.19

Conjecture CompFileWrite2Theorem.2. Subgoals for proof by induction on 'm'
Basis subgoals.
  CompFileWrite2Theorem.2.1  open'(f, 0) == open(f, 0)
  CompFileWrite2Theorem.2.2: open'(f, READ) == open(f, READ)
  CompFileWrite2Theorem.2.3: open'(f, WRITE) == open(f, WRITE)
  CompFileWrite2Theorem.2.4: open'(f, READ_WRITE) == open(f, READ_WRITE)
The induction step is vacuous.


The current conjecture is subgoal CompFileWrite2Theorem.2.1.

The equations cannot be ordered using the current ordering

Subgoal CompFileWrite2Theorem.2.1: open'(f, 0) == open(f, 0)

Subgoal CompFileWrite2Theorem 2.1: open'(f, 0) == open(f, 0)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite2Theorem.2.2.

Subgoal CompFileWrite2Theorem.2.2: open'(f, READ) == open(f, READ)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite2Theorem.2.3.

Subgoal CompFileWrite2Theorem.2.3: open'(f, WRITE) == open(f, WRITE)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite2Theorem.2.4.
```

```
Subgoal CompFileWrite2Theorem.2.4: open'(f, READ_WRITE) == open(f, READ_WRITE)
[] Proved by normalization.

The current conjecture is CompFileWrite2Theorem.2.

Conjecture CompFileWrite2Theorem.2: open'(f, m) == open(f, m)
[] Proved by induction on 'm'.

LP17: prove
Please enter a conjecture to prove, terminated with a '. ' line, or '?' for
help:
  read'(opf, ind, count) = read(opf, ind, count)
  ..


The current conjecture is CompFileWrite2Theorem.4.

The equations cannot be ordered using the current ordering

Conjecture CompFileWrite2Theorem.4:
  read'(opf, ind, count) == read(opf, ind, count)
Proof suspended.

LP24. display deduction File

Deduction rules:

File.2:  when __sel11_name(f1) == __sel11_name(f2),
              __sel5_data(f1) == __sel5_data(f2),
              __sel1_mode(f1) == __sel1_mode(f2)
         yield f1 == f2
File.10: when __sel8_file(r1) == __sel8_file(r2),
              __sel6_data(r1) == __sel6_data(r2),
              __sel2_mode(r1) == __sel2_mode(r2),
              __sel9_fpointer(r1) == __sel9_fpointer(r2)
         yield r1 == r2
File.30: when __sel7_ofile(r3) == __sel7_ofile(r4),
              __sel10_reddata(r3) == __sel10_reddata(r4)
         yield r3 == r4


LP25: instantiate r3 by read'(opf, ind, count), r4 by read(opf, ind, count) in File 30

Deduction rule File.30 has been instantiated to deduction rule File 30 1,
  when
  __sel7_ofile(read'(opf, ind, count)) == __sel7_ofile(read(opf, ind, count)),
       __sel10_reddata(read'(opf, ind, count))
         == __sel10_reddata(read(opf, ind, count))
  yield read'(opf, ind, count) == read(opf, ind, count)

Conjecture CompFileWrite2Theorem.4:
  read'(opf, ind, count) == read(opf, ind, count)
[] Proved by normalization

LP43: declare variable
Please enter variable declarations, terminated with a '..' line, or '?' for
help:
dt:Data
..
```

```
LP44. prove len'(dt)=len(dt)

The current conjecture is lemma CompFileWrite2Theorem.7.

The equations cannot be ordered using the current ordering.

Lemma CompFileWrite2Theorem.7: len'(dt) == len(dt)
Proof suspended.

LP46· resume by induction on dt using Deque.1

Lemma CompFileWrite2Theorem.7: Subgoals for proof by induction on 'dt'
Basis subgoal:
  CompFileWrite2Theorem 7.1: len'(empty) == len(empty)
Induction constant: dtc
Induction hypothesis:
  CompFileWrite2TheoremInductHyp.1: len'(dtc) == len(dtc)
Induction subgoal·
  CompFileWrite2Theorem 7.2. len'(dtc \postcat b) == len(dtc \postcat b)


The current conjecture is subgoal CompFileWrite2Theorem.7 1.

Subgoal CompFileWrite2Theorem 7.1: len'(empty) == len(empty)
[] Proved by normalization.

The current conjecture is subgoal CompFileWrite2Theorem 7.2.

Added hypothesis CompFileWrite2TheoremInductHyp.1 to the system.

Subgoal CompFileWrite2Theorem.7.2: len'(dtc \postcat b) == len(dtc \postcat b)
[] Proved by normalization.

The current conjecture is lemma CompFileWrite2Theorem 7.

Lemma CompFileWrite2Theorem 7· len'(dt) == len(dt)
[] Proved by induction on 'dt'.

LP47. prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help
  toByte(t) = toByte(t)
  .


The current conjecture is CompFileWrite2Theorem.8.

Conjecture CompFileWrite2Theorem.8: toByte(t) == toByte(t)
[] Proved by normalization

LP53· prove
Please enter a conjecture to prove, terminated with a '..' line, or '?' for
help:
  (ind >= 0:Int & ind <= count & toByte(__prefix2(ptr + ind)) =
    __sel10_reddata(read(self', len(toByte(__prefix2(ptr))),
    __sel9_fpointer(self) + (ind *
len(toByte(__prefix2(ptr)))))) & result &
    (ind >= 0:Int & ind <= count & toByte(__prefix2(ptr + ind)) =
    __sel10_reddata(read(self1', len(toByte(__prefix2(ptr))),
    __sel9_fpointer(self) + (ind *
len(toByte(__prefix2(ptr)))))) & result)) =>
```

```
    self' = self1'
  ..

The current conjecture is CompFileWrite2Theorem.9.

The equations cannot be ordered using the current ordering.

Conjecture CompFileWrite2Theorem.9:
  ((ind <= count)
     & (ind <= count)
     & (__sel10_reddata(
          read(self',
               len(toByte(__prefix2(ptr))),
               (len(toByte(__prefix2(ptr))) * ind) + __sel9_fpointer(self)))
          = toByte(__prefix2(ptr + ind)))
     & (__sel10_reddata(
          read(self1',
               len(toByte(__prefix2(ptr))),
               (len(toByte(__prefix2(ptr))) * ind) + __sel9_fpointer(self)))
          = toByte(__prefix2(ptr + ind)))
     & (ind >= 0)
     & (ind >= 0)
     & result
     & result)
    => (self' = self1')
    == true
  Current subgoal:
    ((prefix(
        removePrefix(__sel6_data(self'),
          (len(toByte(__mixfix6(__sel3_locs(ptr), __sel4_idx(ptr)))) * ind)
            + __sel9_fpointer(self)),
        len(toByte(__mixfix6(__sel3_locs(ptr), __sel4_idx(ptr)))))
        = toByte(
            __mixfix6(__sel3_locs(set_idx(ptr, __sel4_idx(ptr) + ind)),
              __sel4_idx(set_idx(ptr, __sel4_idx(ptr) + ind)))))
       & (prefix(
            removePrefix(__sel6_data(self1'),
              (len(toByte(__mixfix6(__sel3_locs(ptr), __sel4_idx(ptr)))) * ind)
                + __sel9_fpointer(self)),
            len(toByte(__mixfix6(__sel3_locs(ptr), __sel4_idx(ptr)))))
            = toByte(
                __mixfix6(__sel3_locs(set_idx(ptr, __sel4_idx(ptr) + ind)),
                  __sel4_idx(set_idx(ptr, __sel4_idx(ptr) + ind))))))
       & ((0 < ind) | (0 = ind))
       & ((ind < count) | (count = ind))
       & result)
      => (self' = self1')
      == true
Proof suspended.

LP54: resume by =>

Conjecture CompFileWrite2Theorem.9: Subgoal for proof of =>
New constants: self'c, ptrc, indc, selfc, self1'c, countc, resultc
Hypothesis:
  CompFileWrite2TheoremImpliesHyp.1:
     (prefix(
       removePrefix(__sel6_data(self'c),
         (indc * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
           + __sel9_fpointer(selfc)),
```

```
        len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
        = toByte(
            __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
              __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc)))))
      & (prefix(
            removePrefix(__sel6_data(selfl'c),
              (indc
                 * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
               + __sel9_fpointer(selfc)),
            len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
            = toByte(
               __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
                 __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc)))))
      & resultc
      & ((0 < indc) | (0 = indc))
      & ((indc < countc) | (countc = indc))
      == true
Subgoal.
  CompFileWrite2Theorem.9 1: self'c = selfl'c == true


The current conjecture is subgoal CompFileWrite2Theorem.9.1.

Added hypothesis CompFileWrite2TheoremImpliesHyp.1 to the system.

Deduction rule lp_and_is_true has been applied to equation
CompFileWrite2TheoremImpliesHyp.1 to yield the following equations, which imply
CompFileWrite2TheoremImpliesHyp.1.
  CompFileWrite2TheoremImpliesHyp.1.1:
     prefix(
       removePrefix(__sel6_data(self'c),
         (indc * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
          + __sel9_fpointer(selfc)),
       len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
       = toByte(
           __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
             __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc))))
       == true
  CompFileWrite2TheoremImpliesHyp.1.2.
     prefix(
       removePrefix(__sel6_data(selfl'c),
         (indc * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
          + __sel9_fpointer(selfc)),
       len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
       = toByte(
           __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
             __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc))))
       == true
  CompFileWrite2TheoremImpliesHyp.1.3: resultc == true
  CompFileWrite2TheoremImpliesHyp.1.4: (0 < indc) | (0 = indc) == true
  CompFileWrite2TheoremImpliesHyp.1.5:
     (indc < countc) | (countc = indc) == true

Deduction rule lp_equals_is_true has been applied to equation
CompFileWrite2TheoremImpliesHyp.1.1 to yield equation
CompFileWrite2TheoremImpliesHyp.1.1.1,
  prefix(
    removePrefix(__sel6_data(self'c),
      (indc * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
       + __sel9_fpointer(selfc)),
    len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
```

```
    ==
    toByte(
      __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
        __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc)))),
which implies CompFileWrite2TheoremImpliesHyp.1.1.

Deduction rule lp_equals_is_true has been applied to equation
CompFileWrite2TheoremImpliesHyp.1.2 to yield equation
CompFileWrite2TheoremImpliesHyp.1.2.1,
  prefix(
    removePrefix(__sel6_data(self1'c),
      (indc * len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
        + __sel9_fpointer(selfc)),
    len(toByte(__mixfix6(__sel3_locs(ptrc), __sel4_idx(ptrc)))))
    ==
    toByte(
      __mixfix6(__sel3_locs(set_idx(ptrc, __sel4_idx(ptrc) + indc)),
        __sel4_idx(set_idx(ptrc, __sel4_idx(ptrc) + indc)))),
which implies CompFileWrite2TheoremImpliesHyp.1.2.

Subgoal CompFileWrite2Theorem.9.1: self'c = self1'c == true
[] Proved by =>

Conjecture CompFileWrite2Theorem.9
((ind <= count)
     & (ind <= count)
     & (__sel10_reddata(
          read(self',
               len(toByte(__prefix2(ptr))),
               (len(toByte(__prefix2(ptr))) * ind) + __sel9_fpointer(self)))
          = toByte(__prefix2(ptr + ind)))
     & (__sel10_reddata(
          read(self1',
               len(toByte(__prefix2(ptr))),
               (len(toByte(__prefix2(ptr))) * ind) + __sel9_fpointer(self)))
          = toByte(__prefix2(ptr + ind)))
     & (ind >= 0)
     & (ind >= 0)
     & result
     & result)
    => (self' = self1')
    == true

[] Proved by =>.
```

Figure 32: LP proofs for completeness of RWFile.lcc Write(char *, size_t) member-function