11-1995

Larch/CORBA: Specifying the Behavior of CORBA-IDL Interfaces

Sankar Gowri Sivaprasad
Iowa State University

Larch/CORBA:

Specifying the Behavior of CORBA-IDL Interfaces

Gowri Sankar Sivaprasad

TR #95-27

November 1995

Keywords: specification, distributed systems, concurrent programming, object-oriented programming, CORBA, IDL, Larch/CORBA, pre-conditions, post-conditions, interfaces, state.

1995 CR Categories: D.1.3 [*Programming Techniques*] Concurrent Programming — distributed programming, CORBA-IDL, documentation; D.1.5 [*Programming Techniques*] Object-oriented Programming — docuementation; D.2.1 [*Software Engineering*] Requirements/Specifications — languages; D.2.7 [*Software Engineering*] Distribution and Maintenance — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, pre- and post- conditions, specification techniques, languages, Larch/CORBA;

Department of Computer Science
226 Atanasoff Hall
Iowa Sate University
Ames, Iowa 50011-1040, USA

Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces

by

Gowri Sankar Sivaprasad

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

Department: Computer Science
Major: Computer Science

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Members of the Committee:

Iowa State University
Ames, Iowa
1995

DEDICATION

$\langle$dedication$\rangle \rightarrow \langle$parents$\rangle$

$\langle$parents$\rangle \rightarrow \langle$mother$\rangle \mid \langle$father$\rangle$

$\langle$mother$\rangle \rightarrow$ **Hema Prasad**

$\langle$father$\rangle \rightarrow$ **Sivaprasad S**

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION 1
CORBA - A Brief Introduction 2
Object Model 3
Object Request Broker 3
Programming Language Mapping 4
CHAPTER 2. INTRODUCTION TO INTERFACE DEFINITION LANGUAGE 5
CHAPTER 3. THE LARCH APPROACH TO FORMAL SPECIFICATION 7
CHAPTER 4. SEQUENTIAL SPECIFICATIONS IN Larch/CORBA 9
An Example 9
Method Specification 11
CHAPTER 5. SPECIFYING CONCURRENCY IN Larch/CORBA 13
Synchronization 14
Atomicity 16
Composition Clause 17
Exceptions and Partial Failures 23
CHAPTER 6. EXAMPLE SPECIFICATIONS IN Larch/CORBA 26
Mutual Exclusion 26
Semaphores 27
Producer Consumer Problem 31
A CD-ROM Scheduler 31
CHAPTER 7. RELATED WORK 35

CHAPTER 8. CONCLUSION AND FUTURE WORK 36
BIBLIOGRAPHY 37
APPENDIX REFERENCE GRAMMAR FOR Larch/CORBA . 39

LIST OF TABLES

Table 5.1: Sets of action sequences generated by the example composition clause 20

LIST OF FIGURES

Figure 2.1: IDL definition of a PrinterQueue 6

Figure 4.1: Larch/CORBA specification of PrinterQueue 10
Figure 4.2: PrinterQueueTrait 11

Figure 5.1: State Diagram for an Operation Execution 15
Figure 5.2: Example of `when` clause 15
Figure 5.3: Composition trait 19
Figure 5.4: ActionSequence trait 20
Figure 5.5: SetActionSequence trait 21
Figure 5.6: ComposeHelpers trait 22
Figure 5.7: Example with a non-atomic operation 24
Figure 5.8: CollectDataTrait 25

Figure 6.1: ThreadTrait 28
Figure 6.2: MutexTrait 28
Figure 6.3: Mutual Exclusion object : A specification 29
Figure 6.4: Semaphores: A specification 30
Figure 6.5: SemaphoreTrait 30
Figure 6.6: ProducerConsumerTrait 31
Figure 6.7: Producer-Consumer problem: A specification 32
Figure 6.8: CDROMTrait 33
Figure 6.9: CD-ROM Scheduler: A Specification 34

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr.Gary Leavens for first suggesting this very interesting and useful project and for numerous useful suggestions and ideas throughout all the drafts (that I stopped counting after some time) of the design.

I would also like to thank the members of the COMS 610GL and COMS 641 classes for the many insights on specification languages that I have gained .

ABSTRACT

The Common Object Request Broker Architecture (CORBA) provides mechanisms for developing heterogeneous, interoperable distributed object systems. In CORBA, the interfaces of objects are specified using the Interface Definition Language (IDL). However, clients and object implementors also need information on the behavior of such objects. Larch/CORBA extends IDL to formally specify the behavior of objects. The language provides mechanisms to specify the concurrency that is inherent in distributed systems. Our goal is to design a language that is usable directly by system designers and programmers. A notable feature of Larch/CORBA is its focus on data rather than on objects in specifying concurrency. It also provides a general mechanism for specifying synchronization. This report presents a preliminary design of Larch/CORBA and a suite of example specifications.

CHAPTER 1. INTRODUCTION

The computer industry, in recent years, has witnessed a proliferation of large networks with many heterogeneous distributed systems. The Common Object Request Broker Architecture (CORBA), defined by the Object Management Group [16], describes an computing model for distributed systems, that is, programs that reside at nodes connected by a network. It provides the mechanisms by which objects[1] communicate among themselves and with their clients. The main functions of CORBA are based on the Object Request Broker (ORB) that transports messages to objects independent of the location (which node the object resides in) of the object. Thus, the ORB hides, from clients and other objects, the actual location of the server object in the distributed environment. The ORB also provides language and operating system transparency between objects and clients and between objects.

CORBA achieves this by defining protocols by which the objects in a distributed environment interact and co-operate. CORBA uses a client-server model, where each server object provides a known set of services that its clients use to build the desired functionality of the application. Objects use *interfaces* to describe the services they offer to the clients. Clients request services from an object using the operations[2] described in this interface. In CORBA, the interface of an object is described using the Interface Definition Language (IDL). Interfaces written using IDL give information about the types and numbers of parameters for operations, but they do not give any information on the functional behavior of the service. This information is not easy to get for the following reasons:

- The source code might contain a lot of implementation details.

[1] Definitions of terms introduced in this section are given in the next section.

[2] By an operation, we mean a service provided by an object i.e., one of the procedures of an object's interface.

- The programming language and the operating system used in the implementation of the object can be different from those familiar to the person writing the client.

- Informal descriptions and documentation are not always precise enough.

We believe that it would be useful to have precise and high-level documentation available for application developers and the people who need to maintain the object implementations. The approach we take in providing this mechanism is to extend the IDL language to be capable of specifying functional behavior. This extended language is called Larch/CORBA. The language provides mechanisms to specify concurrency that is inherent in distributed systems. In specifying concurrency, we follow Lerner [15] and focus on data rather than on the processes themselves. We believe that this encourages modularity and abstraction making the language directly usable to system designers and implementors. Also this maps directly to CORBA's software development model, which depends on a suite of co-operating objects in a distributed environment, providing services to their clients.

It is our hope that Larch/CORBA will eventually evolve into a useful tool in the development of reliable applications in the CORBA framework. It could also serve as a tool in the verification of the implementation and as a formal documentation.

CORBA - A Brief Introduction

This section provides a brief introduction to the Common Object Request Broker Architecture based on [16]. The *Common Request Broker Architecture*, defined by the Object Management Group (OMG), provides mechanisms for objects to transparently[3] make requests and receive responses. It provides interoperability between applications on different machines in heterogeneous distributed environments and interconnects multiple object systems[4].

[3]Two transparencies are involved here: location and implementation transparency

[4]This is part of the version 2 of CORBA specification that has not yet been released at this time

The distributed object management framework helps in thinking of programs as objects in a logically-centralized system. The users of the system are able to use the objects and their services as if they were present locally. CORBA provides the facilities to integrate large numbers of objects present in heterogeneous systems into one single logical object system that hides the implementation and inter-platform interface details.

We give here a brief introduction to the CORBA framework through two topics relevant to this discussion:

- The object model
- The Object Request Broker interface

Object Model

An *object* is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. An *object system* is a collection of objects that decouples the clients (service requesters) from the servers (service providers) by an encapsulating interface. In the object model, the client sends a message to an object to request service. The message identifies the object and the actual parameters.

An *object reference* is an object name that denotes a particular object. An object may be denoted by multiple, distinct object references. Clients refer to objects by their object references.

Each object has an *interface*, which is a description of the set of possible operations that a client may request of the object. The interface of an object is specified using the IDL.

When a service is requested of an object, the object executes some code to provide the service. The code that is executed is called a *method.* The execution of a method is called a *method activation.*

Object Request Broker

One of the aims of CORBA is to provide location and implementation transparency of objects to the clients. The *Object Request Broker(ORB)* is responsible for all the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request. This mechanism is necessary because the objects can be resident anywhere in the distributed system and the client does not have the objects location information (nor should it be required to have the information).

In some situations, some of the interfaces to be used at run-time are unknown. So, in addition to IDL, interfaces can also be specified using the *Interface Repository(IR)*, which permits run-time access to compile-time unknown interfaces.

Programming Language Mapping

Clients see the objects and ORB interfaces through a programming language mapping. Thus clients should be portable across any ORB implementation that supports the language mapping for the client's implementation programming language.

A language mapping specifies the mapping between the programming language's types to the types given in the interface definition and also provides type representations for CORBA's built-in types. Another function of the language mapping is to provide the object representation mechanism, that is the way in which the objects in the system are represented in the programming language. This is an area in which language mappings differ most. Language mappings for C language is provided as part of the CORBA specification[5].

[5] At the time of writing, the mapping for C++ and other languages are in the process of discussion and voting by OMG

CHAPTER 2. INTRODUCTION TO INTERFACE DEFINITION LANGUAGE

The Interface Definition Language(IDL) [16] is the language used by CORBA to describe the interfaces that the objects provide. An interface definition declares the operations an object can perform and their parameters. IDL can also declare attributes of objects, which act as fields or data members of an object. For each attribute defined, "set" and "get" operations are implicitly available to clients to change and fetch the attribute's values. Attributes can be declared as `readonly`, in which case, the set operation is not made available. An inheritance mechanism is available through which interfaces can be inherited. This mechanism provides a way to develop an object-oriented system, even though the implementation language may not support object-oriented features (like inheritance). In such a case, these features get implemented by a mapping to the particular implementation language's features.

As an example, the interface specification of a printer queue, written in IDL, is given in Figure 2.1. The name of the interface is `PrinterQueue`. The interface contains declarations of types and constants. Three operations (in addition to the implicit operations) are defined, namely `enqueue`, `dequeue` and `size`. These are the services that clients can request. The keyword `raises` declares an exception that the operation `enqueue` can signal. Three different parameters mechanisms – `in, out` and `inout` – are permitted. The keyword `in` refers to an value parameter, `out` refers to a result parameter and `inout` refers to a value-result parameter.

An interface can include (by means of the `#include` pre-processor directive) other interfaces and use the attributes and types defined for that interface.

An IDL interface is processed by an *IDL compiler* that is specific to a programming language, which generates the necessary header files and the stub files, in the target language, for the clients and the object implementation to use.

```
interface PrinterQueue {

           const int MAX_QUEUE_SIZE = 20;
           void enqueue (in int id) raises (QUEUE_FULL);
           int dequeue ();
           int size ();
}
```

Figure 2.1: IDL definition of a PrinterQueue

CHAPTER 3. THE LARCH APPROACH TO FORMAL SPECIFICATION

Larch/CORBA is a model-oriented specification language that uses the Larch approach to interface specifications [9]. In this approach, the behavior of operations is specified by Hoare-style pre-conditions and post-conditions [13], together with a specification of what the objects are allowed to change (a frame axiom) and an extension for concurrency [15].

We use the *Larch Shared Language(LSL)* (Chapter 4 in [9]) to describe abstract models. An interface specification consists of a Larch/CORBA specification part and a Larch Shared Language part. The Larch/CORBA part provides the information needed to use the specified object and to write programs that implement it, while the LSL part describes the abstract values of the specification and some vocabulary that is used to manipulate the abstract values that get used in the Larch/CORBA interface. All Larch/CORBA specifications use LSL traits in the same way. LSL plays the same role for other Larch family languages, such as LCL (Chapter 5 in [9]), Larch/C++ [5], LM3 (Chapter 6 in [9]) and Larch/Smalltalk [4]. LSL comes with a set of traits in the form of a LSL Handbook (Appendix A in [9]).

The Larch family of languages support a two-tired, definitional style of formal specification. Larch interface languages encourage the use of abstractions that provides a mechanism for specifying abstract data types. The vocabulary for manipulating the abstract values of the specification is concentrated in the LSL part for important reasons [9]:

- LSL is used by all Larch interface specification languages.
- LSL specifications have simpler semantics than interface specification languages.

- Assertions about properties of the abstract values can be verified using automated tools like the *Larch Prover(LP)* [8].

CHAPTER 4. SEQUENTIAL SPECIFICATIONS IN Larch/CORBA

An Example

An example Larch/CORBA specification, of a printer queue object, is given in Figure 4.1. The associated `PrinterQueueTrait` trait is given in Figure 4.2. This example gives the specification for the IDL interface given in Figure 2.1.

In the Larch/CORBA specification, all the syntax of the IDL specification has been retained while new syntax has been added to specify behavior. The specification proper can be considered in two parts: the header and the operation specifications. The header consists of the `uses` clause and the `initially` clause. The `uses` clause lists the LSL traits used by this specification. We use the `initially` clause to set initial values for the tuple variables used in the trait. We use this clause to specify what the constructor of the object should be initializing. The clause is a predicate that must be true after the constructor has executed. We do not want to specify explicit functions for constructors since object they are typically not called by clients.

The `for` clauses specify the *type-to-sort mapping*. Sorts are names given to the type of abstract values in LSL. The type-to-sort mapping maps the types used in the interface specification to sort names in the traits. Thus it identifies the set of abstract values for each type or object in the specification.

The `PrinterQueueTrait` describes the abstract values of PrinterQueue objects. In the trait, these abstract values have sort `PQ` and we map the `PrinterQueue` object to the sort `PQ` in the uses clause.Thus the `PrinterQueue` object is mapped to the trait `Queue` (Page 171 in [9]) by the combination of the uses clause and the trait included in the `PrinterQueue` trait. So the operations defined in the `Queue` trait can be used on the `PrinterQueue` object.

Three operations have been specified for this object. The behavior of these oper-

```
interface PrinterQueue {

     const int MAX_QUEUE_SIZE 20;
     uses PrinterQueueTrait(PrinterQueue for PQ);
     initially self' = empty;

void enqueue (in int id) raises (QUEUE_FULL) {
     requires true;
     modifies self;
     ensures if len(self^) = MAX_QUEUE_SIZE then
                    raise(QUEUE_FULL) /\ self' = self^
           else
                    self' = append(self^,id);
     }

int dequeue () {
     requires ~isEmpty(self^);
     modifies self;
     ensures self' = tail(self^)
           /\ result = head(self^);
     }

int size () {
     ensures result = len(self^);
     }

}
```

Figure 4.1: Larch/CORBA specification of PrinterQueue

```
PrinterQueueTrait(PQ): trait

     includes Queue(Int for E, PQ for C)
```

Figure 4.2: PrinterQueueTrait

ations are specified by writing, for each operation, a `requires` clause, which gives the pre-condition and an `ensures` clause, which gives the post-condition. The `modifies` clauses lists all the entities that are changed by the execution of the operation. The keyword `self` denotes the object of type PrinterQueue that is the receiver of the message. Two states are defined for each entity: the pre-state and the post-state. The pre-state gives the value of the entity at the time of invocation and the post-state refers to the value at the time of exit from the invocation. The pre-state value of an object in the pre-state is written with a ^ after the entity (as in `self^`) and its post-state value is written with a ' after the entity (as in `self'`). The symbol ~ means a logical negation. The keyword `result` is used to denote the value returned by the operation to the client.

Method Specification

Each method specification has an `ensures` clause. The other clauses are optional. An absent `requires` clauses signifies that the pre-condition is always true, the weakest possible pre-condition. When there is no `modifies` clause given, the method may not change the state of any object. A correct implementation of the interface guarantees the client that if the pre-condition is true when the invocation is made, then the assertions made in the post-condition will be true when the operation terminates. The implementation does not guarantee that the operation will terminate. It only assures that *if* the operation terminates, the post-condition will be true. Thus the specifications imply partial correctness. In the CORBA model, if an operation does not terminate normally, an exception is raised. In case no user-defined exceptions are declared, one of the standard exceptions (See [16] for the list of standard exceptions) defined by the CORBA specification are raised.

The post-condition of the `size` method in Figure 4.1 specifies that the value returned (denoted by the keyword `result`) is given by the predicate `len` from the `Queue` trait. Note that the specification defines a post-state value for `out` parameters. It does not make sense to define a pre-state value for such a parameter. An `inout` parameter has a value in both these states.

CHAPTER 5. SPECIFYING CONCURRENCY IN Larch/CORBA

Systems that comply with the CORBA standards are likely to use concurrency. Concurrency can typically help in enhancing performance and often presents efficient solutions to problems that can be decomposed into parts. These separate computations (either at the same site or different sites) require co-operation from other objects/computations, thus leading to complex interactions.

Specification languages for concurrent systems have been developed before. Some of them focus on processes. For example, the CSP language, designed by Hoare[14], specifies synchronization in terms of the set of allowed traces of all objects. In contrast, Larch/CORBA focuses on data, which seems more appropriate in its role in specifying behavior of interface functions. In this it follows the Generic Concurrent Interface Language (GCIL) [15].

Larch/CORBA's concurrency model is that of a set of sequential processes (clients) invoking operations (requests) on a set of objects that can execute concurrently, in a distributed environment.

The main issues in programming distributed and concurrent systems are [7]:

- Synchronization
- Atomicity
- Exceptions and Partial Failures

In the following, we describe how each of these issues are specified in Larch/CORBA. We also give example specifications (or parts of specifications) to illustrate the ideas.

Synchronization

In specification of sequential programs, an operation is assumed to execute as soon as it is invoked. This may not be the case in concurrent execution of multiple processes. Other operations could be executed between the time an operation is invoked and the time it starts executing [2]. Such interleaved operations may change the state of the system that affects the way an operation executes. For example, consider an operation to update a record from a database. Before this operation can be executed, another operation might have locked that particular record. Then the update operation should be delayed until the record is unlocked. Correctness conditions may dictate that an operation be delayed under certain circumstances. Since an operation's execution may be delayed, we must consider not only the state in which it was invoked, but also the state before its execution. *Synchronization conditions* are used to describe the conditions when an operation should execute. An implementation of a concurrent operation executes in a state where the pre-condition is true and synchronization conditions satisfied. For this purpose we use the *when clause* of [15], which specifies when the operation should execute, if the pre-condition was satisfied. Figure 5.1 shows the state changes at each stage of an operation execution, from invocation to termination. The boxes represent states. In a specification, the pre-state (denoted by ^) refers to the state of the first box, before execution starts. Once the `when` clause is true, pre-state refers to the state of the second box and the post-state (denoted by `'`) is the state of the third box.

Synchronization conditions describe the requirements on when an operation may execute, while the pre-condition represents the client's responsibilities when invoking the operation. With concurrent execution of operations, the state of an object may change between the invocation and execution. It is possible that that this state change causes the pre-condition to be falsified. Thus, for the pre-conditions to be effective, the pre-condition should have only conditions that cannot be falsified between the operation's invocation and execution. Any other condition that needs to be checked before execution should be specified in the `when` clause.

As an example, consider the `dequeue` operation of a concurrent queue given in Figure 5.2 The condition that specifies that the queue should not be empty would

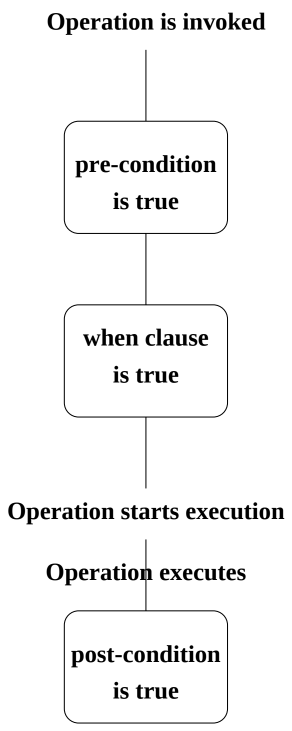

Figure 5.1: State Diagram for an Operation Execution

```
int dequeue() {
      requires true;
      when ~isEmpty(self^);
      modifies self;
      ensures self' = rest(self^) /\ result = first(self^);
}
```

Figure 5.2: Example of `when` clause

have been stated in the pre-condition in a sequential specification. But, in this, concurrent version, it has been moved to the `when` clause, because this condition could be falsified between the operation's invocation and execution. If the `when` clause is false, the `dequeue` operation waits till the queue is not empty and then starts execution.

Atomicity

Larch/CORBA distinguishes between two types of operations: atomic and non-atomic operations. Atomic operations execute till completion once the execution is started, without any visible interleaving with other operations. Thus the state changes made by these atomic operations are visible only at the completion of the operation.

A non-atomic operation consists of a sequence of atomic actions between which actions of other operations may execute. Thus non-atomic operations allow other operations to execute concurrently. (LM3 has a similar model; see Chapter 6 in [9]). When an operation is split into many atomic actions, the operation is said to be *composed* of these actions. An operation that is composed of only one action is, by default, considered an atomic operation and we omit the references to atomic actions inside the specification. The behavior of non-atomic operations are specified by a single pre-condition for the operation and a sequence of `when`, `modifies` and `ensures` clauses, each of which describe the behavior of an atomic action.

In most specifications, atomic operations would suffice. But the very nature of some systems may require the use of non-atomic operations. For example, a multi-user game operating over a network needs to reflect the state of each user in the game environment very frequently. If lengthy operations are executed atomically, then it would not allow the user's states to be updated. Thus these operations have to be split up into atomic actions allowing other operations to interleave. It is sometimes possible to make the actions in an operation themselves as atomic operations, thus eliminating the need for non-atomic operations. But the downside to this is that now the client is responsible for invoking each of these operations, which leads to increased network traffic and performance degradations. Thus systems involving a high degree of user-interaction, reactive systems (systems that have to react to their environment) and real-time systems might require the use of non-atomic operations.

Composition Clause

A *composition clause*, which starts with the keywords `composed of`, is used to specify the order in which the atomic actions of an operation are to be executed. It indicates that any execution of the operation must be equivalent to the execution of the named actions in the given order, possibly interleaved with actions of other operations. The specification of the atomic actions follow the composition clause. Each atomic action specification is prefixed by the keyword `action`. The composition clause is analogous to Path Expressions discussed in operating systems literature [3]. Just as Path Expressions describe the order of process execution, the composition clause describes the order of execution of atomic actions. But the comoposition clause defined here is more powerful than Path Expressions due to the capability to compose arbitrary sets and not just regular sets.

Conceptually, we can imagine the presence of an imaginary scheduler inside each operation. The scheduler has two functions – to decide when each action in the operation executes (synchronization) and to execute the actions in the operation in an order permitted by the composition clause.

Atomic actions themselves are not client callable. They exist only within the operation specification. There is no constraint on the implementation to use atomicity. As long as the changes to the state made by the actions are visible at the appropriate times, the implementation is free to consider various implementations.

In defining the composition clause, we extend the work of [15] and LM3 [9] by allowing user-defined forms of composition to be specified by arbitrary trait functions. This gives the specification developer flexibility in terms of how he/she wants to 'compose' the actions. Since LSL traits can define any computable function, any such function could be used in the composition. We have taken this general purpose approach because specifications might need to have a very complex interaction and/or sequencing of actions

We now define the model we have used in composing actions. To specify sequential execution of actions, a sequence of actions is formed, in which the first action is executed first, then the second, and so on. To specify choices between sequences, a set of sequences is formed, from which the execution must choose. This model is general enough to specify arbitrary executions [12].

The traits, built into Larch/CORBA, for defining the composition clause are given in Figures 5.3, 5.4, 5.5 and 5.6.

Figure 5.5 defines sets of action sequences, and Figure 5.4 defines action sequences. Figure 5.3 defines some syntactic sugar in the form of the trait functions `do` and `\then`. The `List` and `Deque` traits that are used in these traits are given in the LSL Handbook (Pages 173 and 172 in [9] respectively). We give in this report only the traits that we use to define the composition clause and provide basic functionality. These are meant to be extended by users.

Helper functions that can be used in composing complex composition clauses are given in the `ComposeHelpers` trait. These functions are just a sample to show how helper functions can be written and how to use them. For example, using the functions `ifTrue` and `oneFromEach`, we could create a composition clause like:

```
composed of do( oneFromEach(toSequence(ifTrue(i^,action1),action2)),
     \then do(toSequence(ifTrue(j^,action3),ifTrue(k^,action4))) );
```

This clause generates the set of action sequences in Table 5.1, depending on the values of the booleans i,j and k. It also shows how one can generate complex sequences of action executions based on the values of operation parameters. Note that since the composition clause is given before actual operation specification, we require the use of pre-state values for variables used in the trait functions in the composition clause (See Figure 6.3 for an example). Currently we have a few helper functions that operate on action sequences. If there is a need, we could add more functions that manipulate sets of action sequences.

We will now examine a specification using atomicity and the composition clause. Consider an object that collects data from some real-time data source. The system can be stopped and restarted at any time by the user. The specification of this object

```
Composition: trait

   includes SetActionSequence

   introduces
      do: actionSequence -> setActionSequence
      __ \then __ : actionSequence, actionSequence -> setActionSequence
      do: Action -> setActionSequence
      __ \then __ : Action, Action -> setActionSequence
      do: setActionSequence -> setActionSequence
      __ \then __ : setActionSequence, setActionSequence -> setActionSequence
   asserts
   ∀ as, as1: actionSequence, a,a1: Action,
      sas, sas1: setActionSequence

          do(sas) == sas;
          do(as) == toSet(as);
          as \then as1 == toSet(as || as1);
          do(a) == toSet(toSequence(a));
          a \then a1
             == toSet(toSequence(a) || toSequence(a1));
          (as ∈ sas) /\ (as1 ∈ sas1) == (as || as1) ∈ sas \then sas1
```

Figure 5.3: Composition trait

```
ActionSequence(Action,actionSequence):  trait

     includes List(Action for E, actionSequence for C, [__] for {__},
           emptySeq for empty)

     introduces
           skip:  -> Action
           toSequence:  Action -> actionSequence
           toSequence:  Action, Action -> actionSequence
           addtoSequence:  actionSequence, Action -> actionSequence

     asserts
           ∀ a,a1:Action, s:actionSequence

                 toSequence(skip) == emptySeq;
                 toSequence(a) == [a];
                 toSequence(a,a1) == addtoSequence(toSequence(a), a1);
                 addtoSequence(s,a) == s || [a]
     implies ∀ a:  Action
                 toSequence(a,skip) == toSequence(a);
                 toSequence(skip, a) == toSequence(a);
```

Figure 5.4: ActionSequence trait

Table 5.1: Sets of action sequences generated by the example composition clause

i	j	k	Sequence
F	F	F	{[action2]}
T	F	F	{[action1, action2]}
T	T	F	{[action1, action3, action2]}
T	T	T	{[action1, action3, action2, action4]}
F	T	F	{[action3, action2]}
F	T	T	{[action3, action2, action4]}
F	F	T	{[action2, action4]}
T	F	T	{[action1, action2, action4]}

```
SetActionSequence(actionSequence, setActionSequence):  trait

     includes ActionSequence,
           ChoiceSet(actionSequence for E, setActionSequence for C)

     introduces
           toSet:  actionSequence -> setActionSequence

     asserts
           ∀ as:actionSequence, sas:setActionSequence

                 toSet(as) == {as}
```

Figure 5.5: SetActionSequence trait

and the associated trait are given in Figures 5.7 and 5.8. The object is modelled as having an input stream (`istream`) and output stream (`ostream`) of data. For simplicity, we assume the data collected to be integers. Also we assume that the input stream has been filled with the appopriate data. Two tuple variables – `system_started` and `system_stopped` – are used to model the stopping and restarting of the system by the user. We use an `invariant` clause in this specification to state a condition that must always remain true. In this case, the condition is that when the system is in a started state, it is not in a stopped state.

The operation `get_data` collects `n` integers from the input stream and adds them to the output stream. Because of the need to be able to stop and start the system, we cannot have an atomic operation that collects all the `n` pieces of data. We have used the helper function `nTimes` to execute the atomic action `one_step`, that collects one piece of data, `n` times. Thus at the end of each step, other operations could be interleaved that can stop the system. When the system is again restarted, the `when` clause of the `one_step` action is satisfied and the data collection is resumed. Two operations – `start_system` and `stop_system` – have been defined to restart and stop

```
ComposeHelpers:  trait

     includes Composition, ActionSequence, Bool

     introduces
           ifTrue, IfFalse:  Bool, Action -> actionSequence
           oneFromEach:  actionSequence, actionSequence -> actionSequence
           nTimes:  Int, Action -> actionSequence

     asserts
           ∀ a, a1:Action, b:  Bool, i:Int, as,as1:  actionSequence

                 ifTrue(true, a) == [a];
                 ifTrue(false,a) == emptySeq;
                 ifFalse(b, a) == ifTrue(~b,a);
                 nTimes(i,a) == if i >= 1 then [a]
                     || nTimes(i-1,a) else emptySeq;
                 oneFromEach(emptySeq,as) == as;
                 oneFromEach(as,emptySeq) == as;
                 oneFromEach([a] || as, [a1] || as1)
                     == ([a] || [a1]) || (oneFromEach(as,as1))
     implies
           oneFromEach(emptySeq, emptySeq) == emptySeq;
```

Figure 5.6: ComposeHelpers trait

the system respectively.

In the post-condition of the `one_step` action, the pre-state values refer to the state of the system after any previous operations and actions have been completed and the `when` clause has been satisfied. In other words, the notation for refering to pre-state values has been over-loaded. Any pre-state values used in the `requires` clause refers to the state of the system at the time of invocation. Any pre-state values used in the `when` clause refer to the state after any other operations and previous actions have terminated. Any pre-state values used in the post-condition refer to the state after the `when` clause has been satisfied.

Exceptions and Partial Failures

An operation's execution may need to be stopped during execution, when some error or exception condition arises. The specification in Figure 4.1 uses exceptions to notify the client that the printer queue is already full. The keyword `raise` is used to denote that an exception is being raised. Exceptions that an interface can raise are mentioned in the interface header, as part of the IDL syntax.

```
interface Collect_Data {
   uses CollectDataTrait(Collect_Data for CD);
   initially self'.system_stopped = true;
   invariant self.system_started = ~self.system_stopped;
   get_data(in int n) {
      requires n > 0;
      composed of nTimes(n,one_step);
      action one_step {
         when system_started /\ ~system_stopped;
         modifies self;
         ensures self'.ostream = self^.ostream || head(self^.instream)
            /\ self'.instream = tail(self^.instream)
            /\ self'.system_started = self^.system_started
            /\ self'.system_stopped = self^.system_stopped; } }
   start_system() {
      when system_stopped;
      modifies self;
      ensures self' = set_system_started(
         set_system_stopped(self^,false),true); }
   stop_system() {
      when system_started;
      modifies self;
      ensures self' = set_system_stopped(
         set_system_started(self^,false),true); }
```

Figure 5.7: Example with a non-atomic operation

```
CollectDataTrait: trait
     includes List(int for E)
     CD tuple of
           system_started: Bool, system_stopped: Bool,
           instream: C, outstream: C
```

Figure 5.8: CollectDataTrait

CHAPTER 6. EXAMPLE SPECIFICATIONS IN Larch/CORBA

In this section, we give a few example object specifications to illustrate the features and the power of Larch/CORBA.

Mutual Exclusion

Consider the specification of an object that implements mutual exclusion in Figure 6.3. Mutual exclusion (See section 2.1.6 in [17]) is used to protect variables from being acted upon by other processes till the process that is using them is done with them. Thus mutual exclusion can be used to implement critical regions in code. In this example, the system contains multiple threads of execution.

In this specification we introduce the `initially` clause to set initial values for the tuple variables used in the trait. We use this clause to specify what the constructor of the object should be initializing. The clause is a predicate that must be true after the constructor has executed. We do not want to specify explicit functions for constructors since object creation and destruction are handled by the Object Request Broker in CORBA.

The Mutex (See Figure 6.3) interface uses a mutex queue containing a set of `Thread`s that wish to acquire it. The mutex queue holds the identity of threads that wish to acquire it or `NONE`, if no thread is waiting in the queue. `Acquire` and `Release` operations allow a thread to gain hold of a mutex and release it. The `when` clause of `Acquire` prevents a thread from acquiring a mutex when some other thread is holding it. It has to wait until the mutex is released.

We use two traits - `ThreadTrait` and `MutexTrait` (given in Figures 6.1 and 6.2) - to define models for threads and mutex queues. The `ThreadTrait` introduces the `NONE` value as a legal thread identifier. The `MutexTrait` models the queue with the

LSL trait `Queue` (Page 171 in [9]). Note that we do not provide any assertions for the `id` and `thread` operations in the `ThreadTrait`. This is because defining these operations would go into unnecessary detail. `CURRENT` is a Larch/CORBA reserved word used to denote the current thread of execution.

The `Wait` operation is composed of two atomic actions. It allows a thread to temporarily release a mutex (relinquish), wait for some activity by some other thread and then reacquire the mutex (reacquire). A thread may invoke `Wait` only if it holds the mutex. When it relinquishes, it adds itself to the mutex queue. The reacquire action waits until the mutex is available and some other thread has removed it from the mutex queue (using a signal or broadcast operation). Note that even though the thread has been removed from the queue, some other threads might have also been removed from the queue and they could have (re)acquired the mutex before it. Thus it will have to wait till the mutex is available again (and the scheduler schedules it to execute). This is because each action is atomic. This operation is typically used to allow some other thread (say t2) to do something, the effects of which are needed in a thread (say t1) that called `Wait`. The `Signal` operation removes one or more threads from the mutex queue, if the queue is non-empty and the `broadcast` operation removes all of them. The `WhoisHolding` operation can be used to find which thread is holding the mutex variable.

The mutex mechanism could be used to implement the locking model. Locking is commonly used in multi-user databases to protect data from being updated by multiple users at the same time. In an implementation, a mutex variable can exist for each record (or table, based on the granularity of the lock). When a user wants to update a record, the database 'Waits' on the mutex for that record, and when it has acquired the mutex, does the update and then it signals (or broadcasts) the mutex variable, so that it can be used by other users.

Semaphores

The specification of a semaphore variable (See section 2.1.5 in [17] and [6]) is given in in Figure 6.4. The `SemaphoreTrait` (given in Figure 6.5) provides the abstract values of a semaphore object. There are two possible states defined - `available`

```
ThreadTrait(Thread):  trait

     introduces
           NONE: -> Thread
           id, __.Id :  Thread -> int,
           thread :  int -> Thread

     asserts ∀ t:  Thread
           t.Id == id(t)
```

Figure 6.1: ThreadTrait

```
MutexTrait(M): trait

     includes ThreadTrait, Queue(Thread for E, M for C)

     MT tuple of holder:Thread, queue:  M
```

Figure 6.2: MutexTrait

```
interface Mutex {
      uses MutexTrait(Mutex for M);
      initially self'.holder.Id = id(NONE) /\ self'.queue = {};
      void Acquire( void ) {
              requires self^.holder.Id != id(CURRENT);
              modifies self;
              when self^.holder.Id = id(NONE);
              ensures self'.holder.Id = id(CURRENT)
                     /\ CURRENT ∉ self'.queue; }
      void Release( void ) {
              requires self^.holder.Id = id(CURRENT);
              modifies self;
              ensures self'.holder.Id = id(NONE)
                     /\ CURRENT ∉ self'.queue; }
      void Wait( void ) {
              requires self^.holder.Id = id(CURRENT);
              composed of relinquish \then reacquire;
              action relinquish {
                     modifies self;
                     ensures self'.holder.Id = id(NONE)
                     /\ self'.queue = append(self^.queue,CURRENT); }
              action reacquire {
                     when self^.holder.Id = id(NONE);
                     modifies self;
                     ensures self'.holder.Id = id(CURRENT)
                     /\ self'.queue = self^.queue } }
      void Signal( void ) {
              requires self^.holder.Id = id(CURRENT);
              modifies self;
              ensures (self'.queue = {} \/ self'.queue ⊂ self^.queue)
                     /\ self'.holder.Id = id(NONE); }
      void Broadcast( void ) {
              requires self^.holder.Id = id(CURRENT);
              modifies self;
              ensures self'.queue = {} /\ self'.holder.Id = id(NONE); }
      int WhoIsHolding( void ) {
              ensures result = self^.holder.Id; } }
```

Figure 6.3: Mutual Exclusion object : A specification

```
    interface Semaphore {
          uses SemaphoreTrait(Semaphore for S);
          initially self' = available;

          void P() {
                  when self^ = available;
                  modifies self;
                  ensures self' = unavailable;
          }
          void V() {
                  modifies self;
                  ensures self' = available;
          }
    }
```

Figure 6.4: Semaphores: A specification

and `unavailable` through the enumeration in the trait. Two operations are defined on the semaphore variable – the `P` operation and the `V` operation. The `P` operation is used to wait (or halt the execution) till the semaphore becomes available. The `V` operation is used to make the semaphore available to other processes. After a semaphore becomes available and if more than one process is waiting for the semaphore, the scheduler can start the execution of any of such waiting processes.

```
SemaphoreTrait(S): trait

     S enumeration of available, unavailable
```

Figure 6.5: SemaphoreTrait

```
ProdConsTrait(PC): trait

     includes Queue(Int for E)
     PC tuple of buffer:  C, free:  Int
     introduces
          add_to_buffer:  PC, Int -> PC
     asserts ∀ pc:  PC, i:  Int
          add_to_buffer(pc,i) == set_buffer(pc, append(i,pc.buffer))
```

Figure 6.6: ProducerConsumerTrait

Producer Consumer Problem

We consider here the producer-consumer problem (See section 2.1.6 in [17]) that frequently occurs among cooperating processes. In its general form, a set of producer processes supplies messages to a set of consumer processes. They share a limited common pool of space where the messages are placed and removed. Multiple producers can be active at the same time as long there is place in the common pool to place the produced message. If not, producers have to wait till some consumer removes a message from the pool. In the same way, consumers can go concurrently as long as there are messages to be removed or they have to wait for some producer to place a message.

The `ProdConsTrait`, in Figure 6.6, defines the abstract values used by the producer and the consumer. The specification is given in Figure 6.7.

A CD-ROM Scheduler

Imagine a central CD-ROM device, in a networked environment, that services requests for video (or say, documentation) data from multiple clients. In a CORBA environment, the CD-ROM scheduler could be built as an object that provides some specific service to clients (e.g., providing video data upon request). In this section,

```
interface ProducerConsumer {
     const int N = 20;
     uses ProdConsTrait(ProducerConsumer for PC);
     initially self'.free = N /\ self'.buffer = empty;
     void CallProducer(in int m) {
          modifies self;
          when self^.free >= 1;
          ensures self' = add_to_buffer(set_free(self^,
               ((self^.free - 1))),m)
     void CallConsumer(out int m) {
          modifies self;
          when self^.free < N;
          ensures m' = head(self^.buffer)
               /\ self' = set_buffer(set_free(self^,(self^.free - 1)),
                    tail(self^.buffer))
          } }
```

Figure 6.7: Producer-Consumer problem: A specification

```
CDROMTrait(LD): trait

     LD tuple of discpos:  Int, busy:  Bool
```

Figure 6.8: CDROMTrait

we present a simple specification of a scheduler for such a device, in Figures 6.8 and 6.9.

The `Request` operation waits till the device is not `busy`. Then it makes the device busy. In the trait, `discpos` refers to the position of the device's head. The `Request` operation sets the device's head to the destination given by the input parameter `dest`.

The `Release` operation just makes the device not busy. The client of this interface will first call the `Request` operation to take control of the device; then it will use the device for whatever purpose it wants to use it for and finally it calls the `Release` operation to relinquish control of the device.

```
interface CDROM {
uses CDROMTrait(CDROM for LD);
initially self'.busy = false /\ self'.dispos=0;

void Request(in int dest) {
  when ~self^.busy;
  modifies self;
  ensures self' = set_busy(set_discpos(self^,dest),true);
}

void Release() {
  when self^.busy;
  modifies self;
  ensures self' = set_busy(self^,false);
} }
```

Figure 6.9: CD-ROM Scheduler: A Specification

CHAPTER 7. RELATED WORK

Larch/CORBA extends CORBA-IDL to be able to formally specify interface behavior. In this respect, the ADL language [18], designed at Sun Microsystems Labs, is very close in its purpose to Larch/CORBA. But it differs from Larch/CORBA in that ADL's main use is for testing software. ADL specifications are post-condition based and have a well-defined error definition facility. In addition, ADL constructs are designed to allow translation of the formal specifications into natural language documents. In its use in unit-testing software modules, ADL functions as the description of what the software does. In order to enable automated software testing, ADL does not have complex constructs like quantifiers and algebraic specifications[1]. ADL does not meet the goals of Larch/CORBA of being a general purpose specification language for CORBA interfaces due to its lack of complex specification constructs and concurrency support and its main purpose as an aid in testing software.

The Generic Concurrent Interface Language (GCIL) [15] is a Larch interface specification language for specifying concurrent systems. Larch/CORBA and GCIL, being members of the Larch family of interface specification languages, share many common features like the two-tiered model and usage of LSL. Larch/CORBA's concurrency specification features like synchronization (`when` clause) and its focus on data rather than on processes are modeled after GCIL. A GCIL specification describes the objects with which the concurrent processes interact. GCIL can be used in specifying concurrent systems of all kinds. In contrast, Larch/CORBA is tailored to work with IDL interfaces. Another important difference between the two languages is the support for flexible composition of atomic actions using LSL traits and the ability to be able to specify initial conditions on abstract values of interfaces.

[1]The designers of ADL have plans to include these constructs in future.

CHAPTER 8. CONCLUSION AND FUTURE WORK

We have given a preliminary design of a Larch/CORBA specification language which is tailored to describe the behavior of CORBA-IDL interfaces. This language extends the syntax of the IDL specifications and is based on the Larch family of specification languages. It is a hope that the design of Larch/CORBA will serve as a basis for further development of a truly useful tool for formal specification and documentation of CORBA-compliant software.

Some important contributions of this design include the `initially` clause, which allows setting initial values to abstract values and the model for complex composition of atomic actions based on sets of action sequences. This model is powerful enough to be able to express compositions of arbitrary complexity and flexible since it it defined by LSL traits which can be extended easily. This work fills an important need for a facility to formally specify CORBA-IDL interfaces.

Work needs to be done in examining the usefulness (and problems) of specifying large real-world systems using this language. We have not investigated inheritance of interface specifications in this design. Some design changes might be necessary to incorporate this. Practical tools to help build-edit-maintain these specifications are also needed as part of the package. A parser and type-checker should be implemented. Also needed is a careful definition of the semantics of Larch/CORBA. Another area of future research is to examine the implications for Larch/CORBA if IDL is changed or extended.

BIBLIOGRAPHY

[1] H.E.Bal. *The Shared Data Object Model as a Paradigm for Programming Distributed Systems.* PhD thesis. Centrate Huisdrukkerij Vrije Universiteit, Amsterdam. 1989.

[2] Toby Bloom. *Synchronization Mechanisms for Modular Programming Languages.* Technical Report MIT/LCS/TR-211, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA. Jan 1979.

[3] R.H. Campbell, N. Habermann. *The Specification of Process Synchronization by Path Expressions.* Lecture Notes in Computer Science, Vol-16:89-102. Springer-Verlag, NY. 1974.

[4] Yoonsik Cheon and Gary T. Leavens. *The Larch/Smalltalk Interface Specification Language.* ACM Transactions on Software Engineering and Methodology, 3(3):221-253, July 1994.

[5] Yoonsik Cheon and Gary T. Leavens. *A Quick Overview of Larch/C++.* Journal of Object-Oriented Programming, 7(6):39-49, October 1994.

[6] P.J. Denning, T.D. Dennis, and J.A. Brumfield. *Low Contention Semaphores and Ready Lists.* Communications of ACM 8(9):569. September 1965.

[7] G.Goos and J.Hartmanis (eds). *Distributed Systems - Architecture and Implementation.* Lecture Notes in Computer Science, 105. Springer-Verlag, NY. 1981.

[8] Stephen J. Garland and John V. Guttag . *A Guide to LP, the Larch Prover.* Technical Report 82, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA. Dec 1991.

[9] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[10] John V. Guttag and James J. Horning. *Introduction to LCL, a Larch/C interface language.* Technical Report 74, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA. July 1991.

[11] John V.Guttag, James J.Horning, J.M.Wing. *Larch in 5 Easy Pieces.* Technical Report, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA. July 1985.

[12] Wim H. Hesselink. *Programs Recursion and Unbounded Choice.* Cambridge Tracts in Theoretical Computer Science 27. Cambridge University Press. Cambridge CB2 1RP. 1992.

[13] C.A.R. Hoare. *An axiomatic basis for computer programming.* Communications of the ACM, 12(10):576-583, October 1969.

[14] C.A.R. Hoare. *Communicating Sequential Processes.* Communications of ACM 21(8):666-677. August 1978.

[15] Richard Allen Lerner. *Specifying Objects of Concurrent Systems.* PhD Thesis. Carnegie Mellon University, Pittsburgh, PA. CMU-CS-91-131. May 1991.

[16] *The Common Object Request Broker: Architecture and Specification.* Object Management Group, Farmington MI. Document No. 91.12.1, Revision 1.1, Dec. 1991.

[17] A.E.Oldehoeft, Meakawa, Oldehoeft. *Operating Systems - An advanced Approach.* The Benjamin/Cummings Publishing Company, Inc. New York. 1987.

[18] Sriram Shankar and Roger Hayes. *ADL – An Interface Definition Language for Specifying and Testing Software.* Sun Microsystems Laboratories Inc, Mountain View, CA. 1994.

APPENDIX REFERENCE GRAMMAR FOR Larch/CORBA

This section lists the reference grammar of Larch/CORBA in an extended BNF with the following conventions:

• nonterminal symbols are enclosed in angle brackets (e.g. ⟨method-header⟩),
• Larch/CORBA keywords and other terminal symbols are written in **bold** face (e.g., **requires**),
• optional symbols are surrounded by square brackets (e.g., [⟨requires-clause⟩]),
• the notation "..." means that the preceding symbol (or a group of optional symbols) can be repeated zero or more times (e.g., ⟨method-specification⟩ ...). and
• the notation + after a nonterminal means that the preceding nonterminal can occur one or more times. For example ⟨definition⟩$^+$ means that ⟨definition⟩ occurs one or more times.

The lexical conventions are the same as those of CORBA-IDL. For example, ⟨identifier⟩ is an arbitrary long sequence of letters and digits whose first character is a letter. The complete grammar for CORBA-IDL and LSL are not given here for the sake of brevity. Only the relevant parts of these grammars are used. The complete grammar for CORBA-IDL is available in [16].

⟨specification⟩ → ⟨definition⟩$^+$

⟨definition⟩ → ⟨type-dcl⟩ ;
  | ⟨const-dcl⟩ ;
  | ⟨except-dcl⟩ ;
  | ⟨interface⟩
  | ⟨module⟩$^+$

⟨module⟩ → **module** ⟨identifier⟩ (⟨definition⟩)

⟨interface⟩ → ⟨interface-dcl⟩ ⟨forward-dcl⟩

⟨interface-dcl⟩ → ⟨interface-hdr⟩ { ⟨interface-body⟩ }

⟨interface-hdr⟩ → **interface** ⟨identifier⟩ [⟨inheritance-spec⟩]

⟨interface-body⟩ → [⟨attr-dcl⟩ ;] [⟨const-dcl⟩ ;] [⟨type-dcl⟩ ;] [⟨except-dcl⟩ ;]
    ⟨uses-clause⟩ ; ⟨initially-clause⟩] ; [⟨invariant-clause⟩] ; ⟨method-spec⟩

⟨uses-clause⟩ → **uses** ⟨trait-name⟩ **(** [⟨type-to-sort-list⟩] **)** ;

⟨type-to-sort-list⟩ → ⟨type-def-name⟩ **for** ⟨sort-name⟩ [**,** ⟨type-to-sort-list⟩]

⟨initially-clause⟩ → **initially** ⟨predicate⟩

⟨invariant-clause⟩ → **invariant** ⟨predicate⟩

⟨method-spec⟩ → ⟨method-hdr⟩ [{ ⟨method-body⟩ }

⟨method-hdr⟩ → [⟨method-att⟩] ⟨method-type-spec⟩ ⟨method-name⟩
    ⟨parameter-dcl⟩ [⟨raises-expr⟩] [⟨context-expr⟩]

⟨method-body⟩ → [⟨requires-clause⟩] [⟨when-clause⟩] [⟨modifies-clause⟩]
    ⟨ensures-clause⟩
    | [⟨requires-clause⟩] [⟨composition-clause⟩] ⟨action⟩$^+$

⟨composition-clause⟩ → **composed of** ⟨lsl-op-term⟩ ;

⟨action⟩ → **action** ⟨action-name⟩ { [⟨when-clause⟩] [⟨modifies-clause⟩]
    ⟨ensures-clause⟩ }

⟨requires-clause⟩ → **requires** ⟨pre-cond⟩ ;

⟨when-clause⟩ → **when** ⟨predicate⟩ ;

⟨modifies-clause⟩ → **modifies** ⟨store-ref-list⟩ ;

⟨ensures-clause⟩ → **ensures** ⟨post-cond⟩ ;

⟨pre-cond⟩ → ⟨predicate⟩

⟨post-cond⟩ → ⟨predicate⟩

⟨predicate⟩ → ⟨term⟩

⟨term⟩ → **if** ⟨term⟩ **then** ⟨term⟩ **else** ⟨term⟩
    | ⟨logical-term⟩

⟨logical-term⟩ → ⟨logical-term⟩ ⟨logical-opr⟩ ⟨equality-term⟩
    | ⟨equality-term⟩

⟨equality-term⟩ → ⟨lsl-op-term⟩ [⟨eq-opr⟩ ⟨lsl-op-term⟩]
    | ⟨quantifier⟩ ⟨quantifier⟩ . . . **(** ⟨term⟩ **)**

⟨quantifier⟩ → ⟨quantifier-sym⟩ ⟨quantifier-list⟩

⟨quantifier-sym⟩ → ∀ | ∃

⟨quantifier-list⟩ → ⟨identifier⟩ **:** ⟨sort-name⟩ [**,** ⟨identifier⟩ **:** ⟨sort-name⟩] . . .

⟨sort-name⟩ → ⟨identifier⟩
    | ⟨type-def-name⟩

⟨lsl-op-term⟩ → ⟨lsl-op⟩$^+$ ⟨secondary⟩
    | ⟨secondary⟩ [⟨lsl-op⟩ ⟨secondary⟩] . . .
    | ⟨secondary⟩ ⟨lsl-op⟩$^+$

⟨store-ref-list⟩ → ⟨store-ref⟩$^+$

⟨store-ref⟩ → ⟨term⟩

⟨secondary⟩ → ⟨primary⟩
    | [⟨primary⟩] ⟨sc-bracketed⟩ [**:** ⟨sort-name⟩]
    | [⟨primary⟩]

⟨sc-bracketed⟩ → **[** [⟨term-list⟩] **]** **{** [⟨term-list⟩] **}**

⟨term-list⟩ → ⟨term⟩ [⟨term-list⟩]

⟨primary⟩ → **(** ⟨term⟩ **)**
    | ⟨lco-primary⟩

⟨lco-primary⟩ → ⟨literal⟩ | **self** | **result** | **CURRENT**

⟨type-def-name⟩ → ⟨identifier⟩
    | ⟨identifier⟩ **(** ⟨type-def-name⟩ [**,** ⟨type-def-name⟩] . . . **)**

⟨method-name⟩ → ⟨identifier⟩

⟨action-name⟩ → ⟨identifier⟩

⟨sort-name⟩ → ⟨identifier⟩

⟨trait-name⟩ → ⟨identifier⟩