Code Generation for the IOA Language

by

Michael J. Tsai

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Nancy A. Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Certified by .....
Joshua A. Tauber
PhD Candidate
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Code Generation for the IOA Language

by

Michael J. Tsai

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a framework of tools for compiling distributed algorithms written in the IOA language. The target of the compilation is Java code that runs on a network of workstations. The framework includes machinery for matching IOA data types with their implementations and a library of commonly used data type implementations. The interface generator tool produces an auxiliary automaton that connects the algorithm to local sources of I/O. The invocation generator tool assists the user in creating sources of local input. The framework provides two means of resolving nondeterminism: a source-to-source transformer accompanied by user input and a language extension for annotating IOA source files.

Thesis Supervisor: Nancy A. Lynch
Title: NEC Professor of Software Science and Engineering

Thesis Supervisor: Joshua A. Tauber
Title: PhD Candidate

Acknowledgments

I could not have asked for better advisors. Professor Nancy Lynch introduced me to the IOA project three years ago. She has been extremely helpful throughout this project, and her confidence in me has been most encouraging. Josh Tauber, who followed my work most closely, always provided just the right level of guidance and supervision. From discussions of intricate coding details, to high-level design, to how to present my work in writing, he has been an endless source of ideas and constructive criticism. His design of the code generator forms the immediate basis for my work.

Toh Ne Win and Laura Dean participated in many important discussions about IOA and Java and gave valuable feedback as users of my code. Additionally, Toh implemented many crucial classes and answered my random questions. Antonio Ramírez created much of the infrastructure that my code depends upon and helped me learn the IOA codebase. Professor Stephen Garland helped me with the IOA front-end, taught me LSL, and asked thoughtful questions. Atish Dev Nigam and Holly Reimers contributed valuable code, and Atish was a great sounding board for my attempts at explaining the workings of the toolset. Daniel Chvatik commented on drafts of this document.

Other members of LCS have provided suggestions and feedback, including Stanislav Funiak, Professor Michael Ernst, Andrej Bogdanov, and Shien Jin Ong. Professor David Gifford's class helped give me a formal understanding of much of the work I had been doing.

Ultimately, none of this would have been possible without the love and support of my family. It is to them that I dedicate this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The IOA Language and Toolset

Distributed algorithms are becoming increasingly important in modern computer systems, but they are notoriously difficult to design and implement correctly. The Input/Output Automaton model [16] provides a formal way to describe distributed algorithms at a high level of abstraction. Researchers can then reason about them using this concise representation to express properties and theorems. Such analytic work can increase confidence in the correctness of an algorithm.

IOA [9, 10] is a formal language for specifying I/O automata. The IOA toolkit, in development by the Theory of Distributed Systems (TDS) and Networks and Mobile Systems (NMS) groups, is a suite of tools for working with the IOA language. The toolkit is intended as an environment for algorithm and system design using I/O automata. The toolkit has a number of components.

The IOA front-end tool, `ioaCheck`, can check the syntax and static semantics of IOA programs, as well as prettyprint them. Soon it will be able to compose multiple automata into a single primitive automaton; this facilitates hierarchical development of systems. After checking IOA programs the front-end translates them into the *intermediate langauge (IL)*, an s-expression printout of the abstract syntax tree and symbol table.

The IL program may be used as input to a variety of other tools. The `ioa2lsl` [3]

tool converts a restricted class of IOA programs into the Larch Shared Language [11]. The Larch Prover (LP) may then be used to verify theorems about the underlying I/O automata. A prototype translator [7] to TLA+ allows model checking via Lamport's TLC.

The IOA simulator [4], the most developed member of the toolkit, facilitates interactive development of IOA programs by interpreting closed I/O automata on the fly. The user can observe the state of the automaton after each transition and verify that it is operating as expected, while the simulator verifies that the automaton's invariants hold at each step. The simulator supports paired simulation [20], in which two automata are simulated in lock-step. After each transition the simulator verifies that the simulation relation between them holds. Additionally, the simulator can produce execution traces that may be used as input to the Daikon invariant detector [5].

The IOA code generator, currently under development, will compile a restricted class of IOA programs into a standard imperative language. The target of our first implementation is Java. Unlike the simulator, the code generator is not restricted to supporting closed automata. Rather, it can accept an entire system of automata, each of which communicates with the others and exchanges information with the environment. The goal is for the code generator to produce the Java code for each node (automaton) so that the system may be run on a network of workstations. This thesis presents several contributions to the IOA code generator, which are further described below.

1.2 IOA Code Generation

The IOA toolkit may be used to write IOA programs that describe I/O automata and to gain confidence that they are correct. However, putting the algorithm into practice requires coding it up to run on a system of physical machines. This process is time-consuming and error-prone. It is easy to introduce subtle programming errors and quite difficult to detect them. Further, if the IOA specification changes, the code

must be rewritten and the time spent debugging the old code is lost.

The obvious solution is to reduce the chance of errors by minimizing human involvement in the process. This is the primary motivation for the development of the IOA code generator, which is an automated tool that translates IOA programs into Java. The generated code is provably correct, subject to stated assumptions about the correctness of system services and hand-coded data type implementations, and the faithfulness of the code generator's implementation to its design [23].

The output of the code generator is a Java program that represents one node of the distributed computation. Each workstation runs its own copy of the program in a Java virtual machine. The nodes communicate with each other by exchanging messages using the Java bindings [1] for the Message Passing Interface (MPI) [18].

IOA has been designed to be suitable for both verification and code generation. However, these two goals are often at odds; features such as concurrency, nondeterminism, and declarative transition effects simplify verification at the expense of complicating the implementation. The general strategy is for the programmer to design a simple, high-level program that is optimized for verification. He can then successively refine the program into lower-level versions that are more straightforward to implement. Each step of this refinement can be verified, e.g., by proving simulation relations.

The input to the code generator is a program in node-channel form. It is arranged as a collection of automata (one for each node) that communicate via reliable, one-way FIFO channels. Channels connect nodes to each other and to external input and output (modeled as environment automata). The code generator produces Java code that implements the node automata. The implementation of the FIFO channels is the same for each program, and so it is included in a runtime library.

The code generation process consists of a sequence of program transformations. Each transformer reads an IOA program and produces another IOA program in a simpler, more stylized form that is more amenable to code generation. Some of these transformations require input from the programmer. The last transformation produces not another IOA program, but a Java one.

The following is a general outline of the code generation steps. For more details, see Section 1.3 and [23].

- The *interface generator* creates an auxiliary interface automaton for each node automaton. The interface automaton connects the node to external sources of input and output, such as files or the user's console (Figure 1-2).

- The *composer* takes as input the node automaton, the interface automaton, and auxiliary network automata (which are not program-specific). It produces a single primitive automaton for the system comprised of these parts (Figure 1-3).

- The output of the composer is a nondeterministic automaton. The *next action determinator (NAD)* transforms this into an automaton that is *next-action deterministic*. That is, at any given time only one action of the automaton is enabled. In effect, it collects *implicit* nondeterminism throughout the automaton into a single scheduler action where the nondeterminism is made *explicit* (Figure 1-4).

- The user specifies a scheduler for the next-action deterministic automaton, which determines when each action should be enabled and what the values of its parameters should be. This has the effect of removing the nondeterminism so that the automaton becomes *next-state deterministic*.

- Or, instead of the preceding two steps, the user can resolve nondeterminism using annotations in the IOA source file. The code generator can then compile the annotations to Java.

- The final step is for the *code emitter* to generate a Java program from the much-simplified IOA program. The output of the emitter is a subclass of the `ioa.runtime.Automaton` class in the code generator's runtime library.

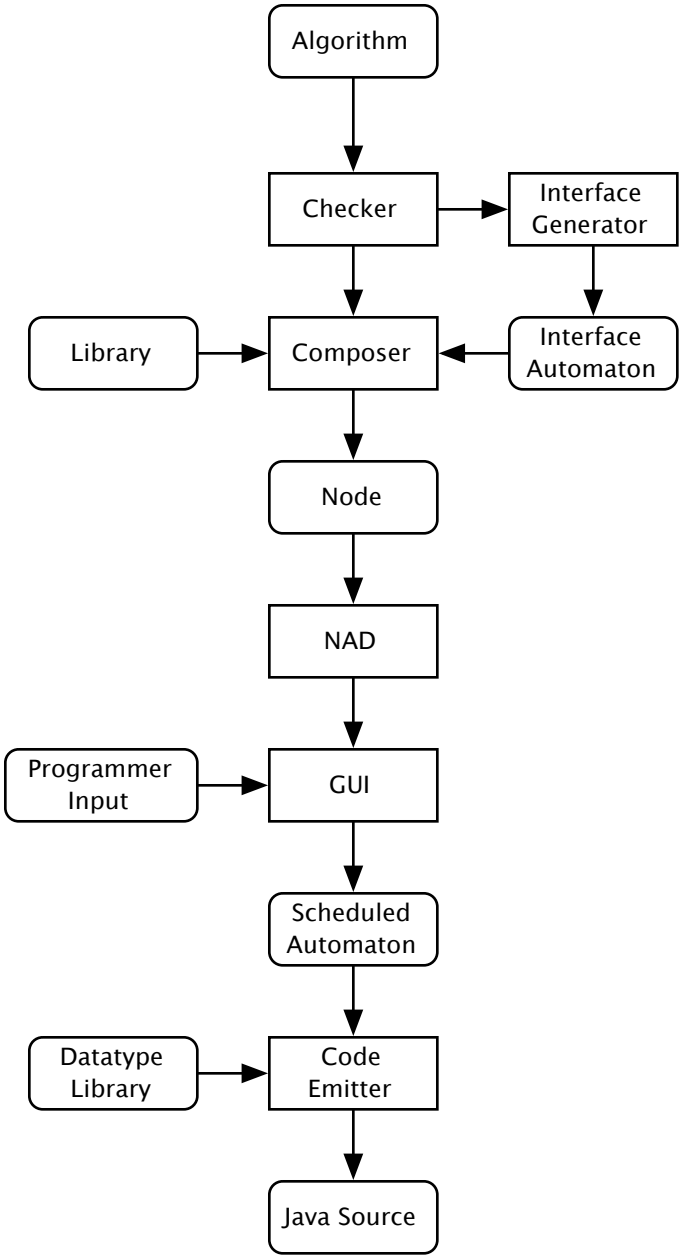

Figure 1-1: Overview of the code generation process when using the NAD to resolve nondeterminism.

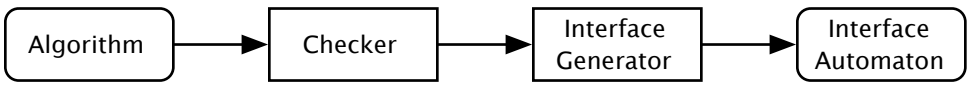

Figure 1-2: Interface automata are created by parsing IOA programs into IL and running the interface generator tool on the IL.

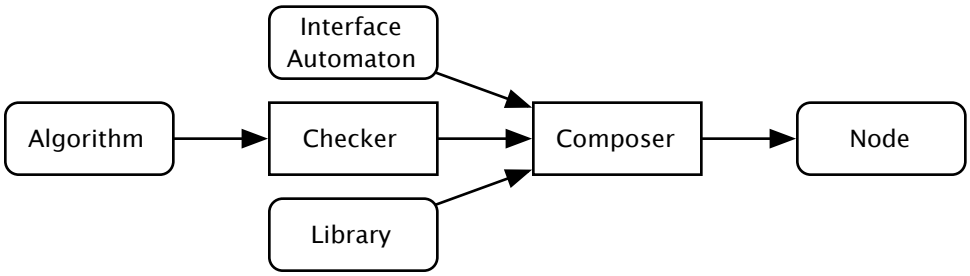

Figure 1-3: Unscheduled node automata are created by composing the original algorithm automata with the interface automata from.

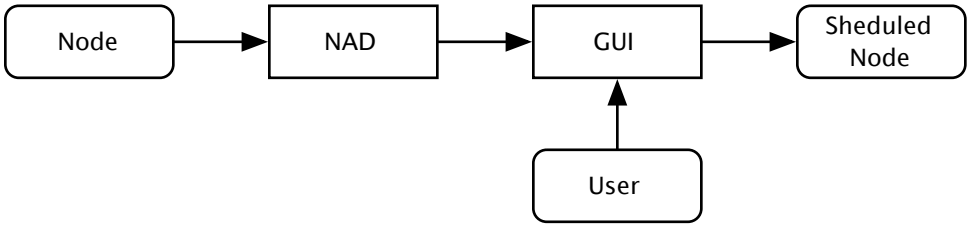

Figure 1-4: Unscheduled node automata are scheduled by running them through the NAD, which makes them next-action deterministic. The user then provides a schedule, making them next-state deterministic.

1.3 Thesis Overview

This section describes the contributions of this thesis to the IOA code generator.

Abstract Data Types

The code emitter's job is to translate a simplified IOA program into Java. In most cases, this translation proceeds in the same way for all IOA programs: an automaton always translates to a class, a transition always translates to a method, and a state variable always translates to a member variable. Data types cannot be translated in this manner because they are extensible. At any time, the user can introduce new data types that are specific to her problem domain.

IOA data types are specified axiomatically using the Larch Shared Language (LSL) [11]. They are implemented in Java, as part of the runtime library. We assume that the data type implementations are correct and that they correspond to the LSL specifications. When the code emitter encounters an operation on a data type,

it must find the corresponding method in the runtime library that implements the operation. This task is handled by the implementation registry. Creating a new data type involves not only implementing it in Java (for inclusion in the runtime library), but also telling the registry which IOA data type and operators the implementation corresponds to. At compile-time, the code emitter consults the registry to look up the implementations of the program nodes it is translating.

With respect to abstract data types, I have:

- Created a new implementation registry and lookup mechanism, based on a prototype by Joshua A. Tauber, which was in turn based on work by Antonio Ramírez-Robredo.
- Added several layers of abstraction to make adding new data type implementations as easy as possible for toolkit users. Registering an operator on a data type now takes one line of code instead of approximately eighteen. This system has been used by Nigam [19] to implement some ADTs that are not built into IOA, and by Win [30] to implement IOA's shorthand data types.
- Completed the implementations of IOA's built-in data types.
- Modified the IOA simulator to use the code generator's registry and data type implementations. Toolkit users can now create a single implementation for each new data type and use it with both the simulator and the code generator.
- Created a mechanism by which the generated code can assign values of the correct type to uninitialized state variables.
- Documented the code generator's ADT architecture and provided a recipe for adding new data types.

Interface Generator

The interface generator takes as input a primitive node automaton and produces for it a customized interface automaton. The interface automaton connects the node

automaton to the environment. It parses input from the console and translates it into input actions on the node automaton. Similarly, it translates output actions on the node automaton into textual output on the console.

Our implementation of the node automaton is single-threaded, and IOA requires that the execution of each transition be atomic. However, just as IOA input actions are non-blocking, console input may arrive at any time. Therefore, an important function of the interface automaton is buffering. The interface automaton runs in two threads of its own, one for input and one for output. It receives input as it arrives and buffers it until the node automaton is ready for the corresponding transition to be invoked.

The interface generator module was envisioned by Tauber [23]. I have provided an implementation, which relies heavily on Ramírez-Robredo's work on the IOA intermediate language [20] and on Reimers's intermediate language unparser [21].

Next Action Determinator

The NAD takes as input a primitive automaton that is the composition of the node automaton and the interface and network automata. It produces an automaton that is next-action deterministic, that is, where only one action is enabled at a given time.

IOA programs contain explicit nondeterminism in the form of **choose** statements and implicit nondeterminism in that transitions may happen in any order and with any parameters that satisfy their preconditions. The NAD makes the implicit nondeterminism explicit and collects it into a single scheduler transition. It assigns each transition a unique index and adds a program counter state variable to the automaton. The precondition of each transition is modified so that it is only enabled when the program counter is equal to the transition's index. The scheduler transition is responsible for setting the value of the program counter and determining parameter values that satisfy the precondition of the enabled transition.

The NAD transformation is due to Vaziri, Tauber, and Lynch [28]. I have provided an implementation, in the form of a source-to-source transformer. Vaziri [29] has shown that the semantic transformation of the I/O automata and the syntactic

transformation produced by the code generator's NAD module are equivalent.

Compiling the Nondeterminism Resolution Language

An alternative method of resolving nondeterminism is to annotate the IOA program using the nondeterminism resolution language that Ramírez-Robredo developed for the simulator. The annotations allow the user to specify a schedule and also to determine the values of **choose** statements. I have extended the code emitter to compile the annotations to Java. This approach to resolving nondeterminism has many practical advantages, and I expect that users will prefer it to using the NAD tool.

Invocations and Runtime I/O

The interface automaton's job is to translate input from the local environment into input actions and to translate output actions into output to the local environment. I have defined an s-expression representation for invocations of IOA actions. The interface automaton reads a file consisting of a list of these invocations and causes the corresponding input actions to be executed. When an output invocation is executed, the interface automaton creates an invocation s-expression and appends it to an output file. Because invocations are tedious to create by hand, I have created a convenience tool called the invocation generator that creates them from **fire** statements written in the nondeterminism resolution language.

Other

Other contributions in this thesis include:

- Work on the IOA front-end—in particular, implementation of the intermediate language translator and parts of the prettyprinter.
- Enhancements to the intermediate language classes, including improved parsing, printing, and type checking.

- Enhancements to the prototype code emitter to handle transition parameters, preconditions, **where** clauses, and other features related to data types.
- Implementations of the auxiliary interface automata that link the node automaton to the console.
- Enhancements to the data type implementations so that they can be transferred across the network, and a simple experiment that demonstrates how to use mpiJava with the IOA toolkit.
- A prototype graphical user interface from which the user can edit IOA programs and run some of the tools that are needed for the code generation process.
- A regression testing framework for the IOA toolkit, along with a suite of unit tests and regression tests for the modules described in this document.

1.4 Writing Conventions

This thesis includes code fragments and program examples written in several different programming languages. The following typesetting conventions are used to help the reader:

- Sans-serif type is used for IOA and LSL, with language keywords in **bold**.
- `Constant width` type is used for Java, with language keywords *`slanted`*.
- `Constant width` type is also used for grammars, filenames, command names, URLs, program output, and strings.
- Grammars are expressed in BNF, and terminals are enclosed in quotation marks.

In addition, several potentially vague terms are used as follows:

User a person using the IOA toolset to develop and verify algorithms. Users program in IOA and LSL.

Programmer a person developing the IOA toolset itself. Programmers program in Java and PolyJ. Users become programmers when they extend the IOA toolset to support additional abstract datatypes.

Algorithm automaton the original automaton that the user begins with and that he may also use with the simulator or LP. During the code generation process, the algorithm automaton will be composed with other auxiliary automata.

Compile-time when the code emitter translates the final automaton (a composition of the algorithm automaton and auxiliary automata) from IOA into Java.

Runtime when the generated Java code runs on a network of workstations.

Node a networked workstation that carries out one part of the distributed computation by running the generated Java code.

Chapter 2

Abstract Data Types

2.1 Overview

2.1.1 Code Generation

When the IOA code generator converts an IOA program into runnable Java code, it first parses the program, which has already been translated from IOA into the intermediate language (IL) by the front-end. The result of this parse is an abstract syntax tree with nodes for each element of the IOA program (e.g. transitions, state variables, and operators). The source syntax tree consists of objects from the `ioa.codegen.source.java` package.

The next step is to translate the source syntax tree, node by node, into the target syntax tree. The target tree consists of objects from the `ioa.codegen.target.java` package and has nodes that correspond to Java language elements (e.g. classes, variables, and methods). In the final step, the code generator walks the target syntax tree, asking each node to emit itself as a string. The result of emitting the whole target tree is a Java program that can be compiled and run, with the help of classes in the `ioa.runtime.adt` and `ioa.runtime.io` packages. These runtime packages are self-contained; no other part of the code generator is needed to run the generated code.

The relation between the different parts of the code generator is summarized by

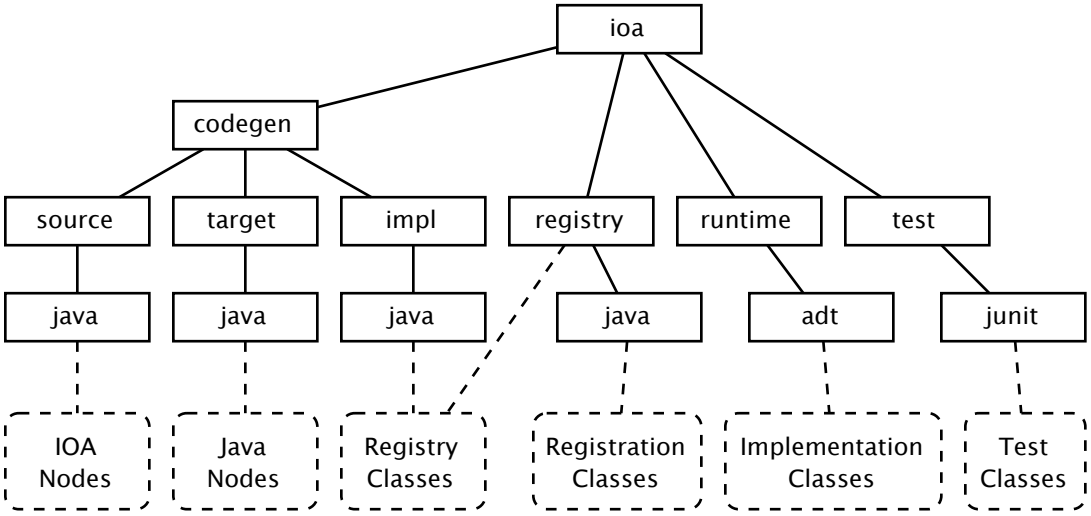

Figure 2-1: Code Generation Package Structure

the package diagram in Figure 2-1.

The brunt of the code generator's work is in translating the nodes of the source tree into nodes of the target tree. In most cases, this translation proceeds in the same way for all IOA programs: an automaton always translates to a class, a transition always translates to a method, and a state variable always translates to a member variable. These translations can be hard-coded into the nodes of the source syntax tree. However, data types must be handled differently, and we will talk about them now.

2.1.2 Data Types

IOA data types are divided into two categories: *sorts* and *sort constructors*. Sorts are simple data types such as integers (Int), real numbers (Real), and booleans (Bool). Sort constructors are compound data types that are parameterized by other sorts or sort constructors. Examples include sequences of integers (Seq[Int]) and mappings from strings to sequences of integers (Map[String, Seq[Int]]). Most of the discussion in this thesis applies to both sorts and sort constructors; therefore, for brevity, I will use "sort" to mean "sort or sort constructor" and clearly indicate sections that apply

only to one category of data types.

Data types cannot be translated in the fixed manner of Section 2.1.1 because they are extensible. At any time, the programmer can add new data types by specifying them in the Larch Shared Language (LSL) [11] and implementing them in Java. Doing so should not require modifying the core of the code generator.

Each IOA sort is implemented by a Java class, and each operator is implemented by a method on that class. At compile time, one of the code generator's jobs is to map IOA sorts and operators to their Java implementations, so that it can create the proper nodes in the target syntax tree. Much of the work in adding support for new data types involves telling the code generator about this correspondence.

The classes involved in generating code for abstract data types (ADTs) may be divided into three categories:

- *Implementation Classes*, which implement sorts and their operators. These are written by the user, one per data type, and live in `ioa.runtime.adt`. Of the three kinds of classes, this is the only one that is needed at runtime (i.e. when compiling or running the generated code).

- *Registry Classes*, which maintain a mapping between IOA sorts (and operators) and the Java implementation classes (and methods) that implement them. These are built into the code generator.

- *Registration Classes*, which interface between the implementation classes and the registry classes so that each group is isolated from the other. These are written by the user, one per implementation class. They implement the `Registrable` interface. Registration classes that are part of the standard IOA toolkit distribution belong to the `ioa.registry.java` package. If a user extends the code generator to support additional data types, their registration classes may belong to one of the user's packages.

The relation between these categories and the package structure is shown in Figure 2-1. The dependencies between them are shown in Figure 2-2; in particular, note that registration classes decouple the implementation classes from the registry.

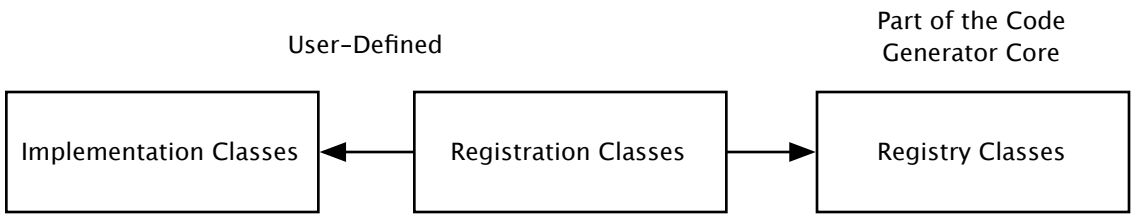

Figure 2-2: The three types of ADT classes and their dependencies. Registration classes call methods of the registry classes and depend on the names of the methods in their corresponding implementation class. The registry classes and implementation classes are independent of one another.

2.1.3 Chapter Roadmap

The remainder of this chapter describes how the code generator's ADT classes are designed and how they may be extended to support additional sorts. Section 2.2 tells how IOA sorts are implemented in Java, Section 2.3 explains how the code generator uses the registry to map sorts and operators to their implementations, and Section 2.4 describes how to install new implementation classes into the registry. Section 2.5 explains the testing architecture for verifying implementation classes, registration classes, and the correspondence between them. Section 2.6 gives a recipe for writing new ADTs. Section 2.7 explains how the work described here has been reused in the IOA simulator. Sections 2.8.1 and 2.8.2 explain the data types that have already been implemented, and Appendix A contains complete examples of the files needed to add support for new data types and examples of the generated Java code.

2.2 Implementation Classes

The code generator supports the standard sorts defined in the IOA manual [9]: Array, Bool, Char, Int, Map, Mset, Nat, Real, Seq, Set, and String; as well as the shorthand sorts **enumeration**, **tuple**, and **union**. Each sort is implemented by a Java class called an *implementation class* that belongs to the `ioa.runtime.adt` package. By convention, the name of the implementation class is the name of the IOA sort followed by "Sort". Thus, the IOA boolean sort, Bool, is implemented by `BoolSort`; and the IOA multiset

sort, Mset, is implemented by `MsetSort`. For brevity, in this section I will sometimes refer to the implementation class as "the ADT."

Each IOA operator is associated with a single sort that introduces it and is implemented by a static method in the implementation class of the introducing sort. For instance, the $\_\_+\_\_$: Int , Int $\rightarrow$ Int addition operator on integers is introduced by Int and implemented by `IntSort.add()`. When the code generator generates code for a program, it translates each operator application into a static method invocation. Thus, the static methods form the interface between the implementation class and the rest of the generated code.

2.2.1 ADTs Extend `ioa.runtime.adt.ADT`

Implementation classes extend the `ioa.runtime.adt.ADT` abstract class. They must override the non-static `equals()` method inherited from `java.lang.Object`, so that it checks for value equality instead of reference equality.

The `ADT` abstract class provides implementations for two operators that are common to all IOA data types: equality ($\_\_=\_\_$) and inequality ($\_\_\neq\_\_$). These operators are implemented by static methods in `ADT`. `equals()` and `notEquals()` take two ADTs (classes that extend `ADT`) as parameters and return `BoolSort`s that indicate whether the ADTs are equal or unequal. The implementations for these operators are automatically installed into the registry; there is no need to mention them in the registration class (see Section 2.4).

2.2.2 ADTs are Immutable

ADTs are immutable[1]. Each ADT overrides `equals()` so that it returns *`true`* for objects that represent the same *value*, even though they may not be the same *object*. Because they override `equals()`, ADTs must also override `hashCode()` so that it satisfies the Java language's `hashCode()` contract [22]:

[1]Thus, all of the container ADTs are slow because insertion and deletion require an amount of copying that is linear in the number of contained elements. Future work could optimize away the unnecessary copying.

> ...If two objects are equal according to the `equals()` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.

If this clause of the contract is violated, collection data types that use hash tables will fail. They will not be able to find objects that they contain, making it impossible to test for membership or remove objects.

Since immutable ADTs properly override `hashCode()`, container ADTs may be implemented using hash tables. For instance, `MapSort` uses a hash table to maintain a mapping, and `MsetSort` uses one to map elements to their multiplicities.

In addition, all ADTs should override `toString()` to "unparse" themselves; this is important for readable output from the simulator [4, 20, 5] and also when debugging the code generator.

2.2.3 Static Methods Implement Operators

The implementation class contains a public static method, also called a *code generation method*, for each of the sort's operators. Making the methods static simplifies code generation by making the syntax of the generated code more regular. However, because static method calls are verbose and unnatural for writing Java data types, all but the simplest implementation classes define two sets of methods: *instance methods* implement the operators on the sort, and *public static methods* are called by the generated code.

The public static methods are just wrappers for the instance methods, so it is simple to implement them. Maintaining these two sets of methods is a little extra work compared to implementing everything using the public static methods; however, the instance methods are useful to have when hand-coding programs elsewhere in the toolkit.

The parameters and return values (if any) of the static methods are instances of `ADT`. If the exact type of a parameter is known, then it is specified in the method

signature; otherwise, it is labelled `ADT`[2] to indicate that it is known to be an implementation class. The order of the method's parameters is the same as the order of the operands in the domain of the IOA operator. This means that in most cases the first parameter happens to be an instance of the ADT that defines the method. For example, the prototype of the method for the indexing operator on mappings __[__]: Map[D, R], D → R is *public static* `ADT get(MapSort map, ADT key)`. Its implementation simply calls the `MapSort` instance method *public* `ADT get(ADT key)`.

2.2.4 Return Value Casting

Java is a statically typed language, and a Java program will only compile if the compiler can verify that methods are called with parameters of the correct types. Container data types are usually written to contain `Object`s, so after taking an `Object` out of a container, one must downcast it (cast it to a more specific type) before using it. This is a general problem with Java containers such as `Vector` and `Hashtable`, and it affects implementation classes in the same way.

To deal with this, whenever the code generator emits a method call, it also emits a downcast for the return value. For instance, the generated code for an IOA term such as head(aSeqOfInt) might be `((IntSort)SeqSort.head(aSeqOfInt_v0))`. The code generator handles this automatically (see Section 2.3.7). The implementation class writer can assume both that parameters to his classes' methods will be downcast so that they satisfy the method signatures and that non-specific return types, such as `ADT`, will not pose problems for code using his methods.

2.2.5 Arbitrary Initialization and Parameterizations

The IOA language allows state variables to be initialized explicitly or arbitrarily. Explicitly initialized variables are assigned values by the programmer. For instance, an automaton's states section might include the line

```
a: Array[Int, Bool] := constant(true)
```

[2]Or `ComparableADT`; see Section 2.2.7.

which initializes a to an array where the value at each integer index is the boolean constant **true**. Alternatively, the programmer may write a: Array[Int, Bool]. This indicates that the variable is arbitrarily initialized, and IOA allows the elements of a to take on any values so long as they are of the correct sort. In this case, the code generator must initialize a before firing any of the automaton's transitions.

When the code generator needs to create an initial value for a variable, it first finds the implementation class for the variable's sort and calls its method

```
public static ADT construct(Parameterization p)
```

which is charged with creating and returning "some" instance of the implementation class. The `Parameterization` parameter provides the implementation class with information about the subsorts of the sort constructor that it is implementing.

In the base case, `construct()` is being called on the implementation class of a simple sort. In this case, the `Parameterization` carries no useful information and is ignored. `construct()` returns a suitable default value.

In the recursive case, `construct()` is being called on the implementation class of a compound sort. Typically, constructing the compound sort will require recursively constructing instances of the subsorts. For instance, because each index of an array must have a value, an array must construct at least one instance of its element type subsort. It can then use this to initialize itself. The `Parameterization` contains all the information about the subsorts, so the array can call its `ADT constructSubsort(int)` method to create an instance of one of the subsorts.

In the case of arrays, `construct()` looks like:

```
public static ADT construct(Parameterization p)
{
    ADT elementValue = p.constructSubsort(p.nSubsorts() - 1);
    return constant(elementValue);
}
```

It simply creates an instance of the element subsort[3] and creates an array where all the slots have the same value.

Since the generated code contains `Parameterization` objects, `Parameterization` must be part of the code generator's self-contained runtime package. (See Figure 2-1.) The registry classes are *not* part of the runtime, so `Parameterization` may not use them when it recursively constructs subsorts. Therefore, the code generator looks up all the needed implementation classes at compile-time and stores the results in the `Parameterization`. Each `Parameterization` stores the implementation class and `Parameterization` for each of its subsorts.

Ultimately, the purpose of a `Parameterization` is to provide the information necessary for the ADT implementation to construct a value of the proper type. When the code generator translates a state variable, it creates a member variable in the target syntax tree. If the IOA program initialized the state variable, it translates the IOA initializing term into a Java one. Otherwise, it creates an `InitialValue` node in the target syntax tree. An `InitialValue` is a term that stores the class that will generate the value, as well as a `Parameterization` containing the needed subsort information.

Emitting an `InitialValue` creates a call to the `construct(Parameterization)` method of the implementation class for the value's type, e.g. `ArraySort.construct()` for the above example. The parameter of the method is generated by emitting the `InitialValue`'s stored `Parameterization` object. The return value of `construct()` is cast (from `ADT`) to the appropriate type.

The result of emitting a `Parameterization` is a Java fragment that reconstitutes the `Parameterization`. Again using the above example, the generated code for `a: Array[Int, Bool]` is:

```
ArraySort a_v0 =
        (ArraySort)ArraySort.construct(
                        new Parameterization(
                             new Class[]
                              {ioa.runtime.adt.IntSort.class,
                               ioa.runtime.adt.BoolSort.class},
                             new Parameterization[]
```

[3]The `nSubsorts() - 1` is necessary because arrays can have two or three subsorts, depending on whether they are one- or two-dimensional; the element subsort is always the last one.

```
                {new Parameterization(),
                 new Parameterization()}));
```

The parameters to the outermost `Parameterization` constructor are an array of implementation classes for the subsorts and an array of `Parameterization`s for them. The innermost `Parameterization` constructors take no arguments because `Int` and `Bool` have no subsorts.

The astute reader will have noticed that `Parameterization` is seemingly a runtime class that knows how to emit itself. This is impossible because knowledge about emitting is confined to the code generator's internals and is not part of the runtime. There are, in fact, two `Parameterization` classes: `ioa.runtime.adt.Parameterization` and `ioa.codegen.target.java.Parameterization`. The former contains the functionality needed at runtime, and the latter (which is a subclass) handles the emitting. Creation of `Parameterization`s is handled by the registry classes' factory (`ImplFactory`), which in the case of the code generator always creates emittable `Parameterization`s.

2.2.6 Inheritance

ADTs can inherit method implementations from their superclass. For instance, `equals()` and `notEquals()` are implemented in `ADT`, and the other ADTs inherit these implementations. When `BoolSort.equals()` is called, `ADT.equals()` is invoked. Inheriting observer methods in this manner will also work for user-defined methods.

In contrast to observer methods, producer methods return new instances of the data type. For instance, `SeqSort.append()` returns a new `SeqSort` based on the original and the parameter. If a subclass of `SeqSort` does not override `append()`, then `append()` will continue to return `SeqSort`s; it will not return instances of the subclass. This is clearly not acceptable because then calling `append()` demotes the subclass to a `SeqSort`, and it *cannot be promoted by casting* because it actually was created as a `SeqSort` and nothing more.

On the other hand, if the subclass is written to override `append()`, then it has gained little from inheritance: just to add a method or modify an existing one, every

one of the ADT's producers must be rewritten to return instances of the subclass. And if a producer is added to the base class, then all the derived classes will break. In summary, inheritance does not work as well as we would like, but the limitations lie with immutability rather than with our design of the ADT classes.

2.2.7 Comparing ADTs

Container ADTs such as priority queues require that the elements they contain be totally ordered. To support this, element ADTs that support comparison (such as Int, Real, and String) extend `ioa.runtime.adt.ComparableADT` instead of `ADT`. To do this, they must provide implementations of the following method:

```
public int compareTo(Object object)
```

The parameter is assumed to be a `ComparableADT`, but it is declared as an `Object` so that `ComparableADT` can implement `java.lang.Comparable`. This lets the ADT implementation use Java containers such as trees and utility functions such as `sort()`. The return value of `compareTo()` should be zero if the two ADTs are equal, negative if *this* is less than `object`, and positive if the reverse is true. In ADTs such as `IntSort` that define comparison operators, the comparison operators are implemented in terms of `compareTo()`.

Operators on ordered data types are implemented in the normal way, except that generic data type parameters are declared as `ComparableADT`s instead of `ADT`s. For example, the priority queue implementation contains this method:

```
public static PQSort add(ComparableADT a, PQSort p)
```

Although the priority queue trait assumes that the element sort is totally ordered, the IOA checker does not prevent the user from creating priority queues of uncomparable types. In the case of the code generator, such errors will be caught at compile-time when the Java compiler complains that an `ADT` has been passed to a method that requires a `ComparableADT`. The same code when run through the simulator results in a `SimException` that reports an illegal method argument.

2.2.8 Exceptions

The static code generation methods should signal errors and representation violations by throwing `RepExceptions` or subclasses of `RepException`. These unchecked exceptions will be caught by the runtime system. Examples of situations in which it is appropriate to throw a `RepException` are when the code tries to:

- divide by zero
- find the predecessor of the natural number zero
- take the head or tail of an empty sequence
- pop an empty stack
- access a non-current value of a union

2.3 Registry Classes

When the code generator translates an IOA term into Java, it must match IOA sorts and operators to Java classes and methods. To do this, it uses the registry classes, which maintain a mapping between IOA objects and their Java implementations. The principal class that maintains this mapping is `ConstrImplRegistry`, whose name stands for "Constructor Implementation Registry." I will refer to it simply as "the registry."

The contents of the registry are diagrammed in Figure 2-3.

The registry is used in two phases. In the first phase, the registration classes install their implementation classes into the registry. This process is described in Section 2.4. In the second phase, the code generator uses the registry to look up the implementations of the IOA objects that it needs to emit. These two phases are diagrammed in Figure 2-4 and are described below.

To look up a sort or operator, the code generator calls `getImpl()` on the registry. This method takes a single parameter, an intermediate language `ioa.il.Sort`

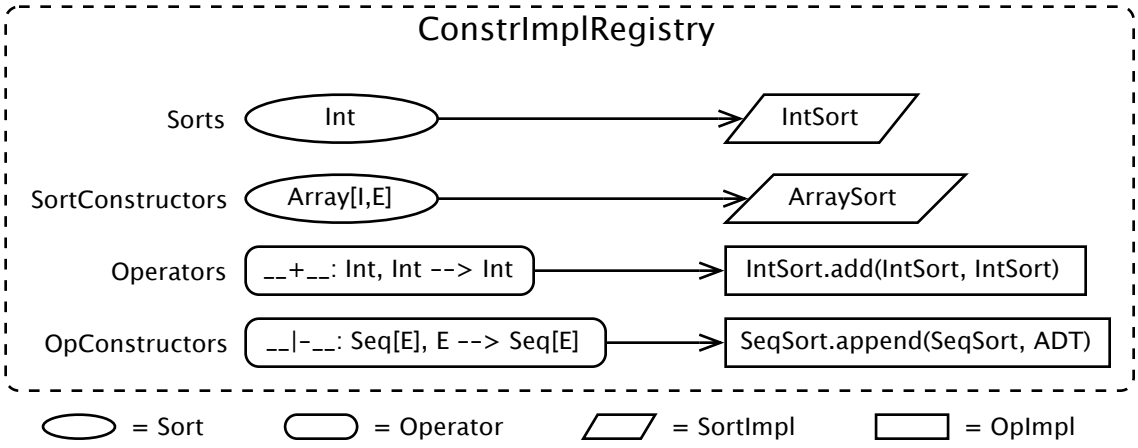

Figure 2-3: Abstract Contents of the Registry

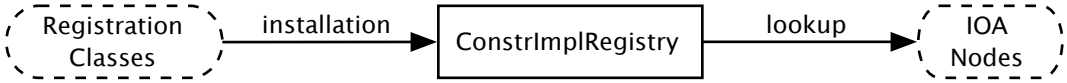

Figure 2-4: Information Flow and the Registry

or `ioa.il.Operator`, and returns an instance of `ioa.registry.SortImpl` or `ioa.registry.OpImpl`, respectively. How the lookup process works and what kinds of `SortImpl`s and `OpImpl`s (which are both subtypes of `Impl`, a marker interface for implementations) are returned depend on whether the sorts are simple or compound.

2.3.1 Looking up Simple Sorts and Operators

For simple sorts and operators, the registry maintains two tables. The *sort table* maps `Sort` keys to `SortImpl`s, and the *operator table* maps `Operator` keys to `OpImpl`s. Keys are simply structured `String` representations that include the name of the sort or operator and (recursively) all of its subsorts. The syntax of keys is an implementation detail that is subject to change; they are created using the `makeOpKey()` and `makeSortKey()` methods of `ConstrImplRegistry`.

The registry's `installSortImpl()` and `installOpImpl()` methods let one add mappings to the tables, and the `getImpl(Sort)` and `getImpl(Operator)` methods use them to look up `SortImpl` and `OpImpl` objects for simple sorts and operators.

When looking up a simple sort, the registry returns a `SortImpl` that is an instance

of `ioa.codegen.target.java.Class`. The `Class` object knows the name of the class that implements the sort (e.g., `ioa.runtime.adt.SeqSort`). The code generator calls upon it to emit these pieces of information when the time is right.

Looking up simple operators works in the same way. The registry returns an `OpImpl` that is an instance of `ioa.codegen.target.java.Operator`. The `Operator` is a node in the target syntax tree, and it knows which method (e.g., `SeqSort.head()`) implements the IOA operator that it represents. It also remembers which `ioa.il.Operator` object it is supposed to be implementing. When the code generator asks the `Operator` to emit itself, it passes the `Operator` a list of actual parameters. The `Operator` emits the name of the method (e.g., `SeqSort.head`) followed by the list of parameters (delimited by commas and enclosed by parentheses). It emits a cast of the return value of the method to the implementation class of the sort that the IOA operator returns. This lets one, for instance, take a `StringSort` out of a `SeqSort` and assign it to a variable of type `StringSort`, even though `SeqSort.head()` returns a vanilla `ADT`.

2.3.2 Looking up Compound Sorts and Operators

For compound sorts and operators, the registry maintains two more tables. The *sort constructor table* maps `String`s to chains of `SortConstructors`, and the *operator constructor table* maps `String`s to chains of `OpConstructor`s. Unlike their `Impl` counterparts, `SortConstructor` and `OpConstructor` are not nodes of the target syntax tree. Instead, they are intermediary objects that know how to *construct* `Class`es and `Operator`s. They represent IOA sort and operator constructors and extend `ioa.registry.Constructor`.

The keys of the sort constructor table and operator constructor tables are shallow IOA names (omitting subsorts) for sorts and operators (e.g., `Map` and `__|-__`). The values are chains of constructor objects that share the same shallow name. When looking up a compound `Sort` or `Operator`, the registry uses the corresponding table to find the appropriate chain of constructors. If the code generator has an implementation for the `Sort` or `Operator`, then one of the `Constructor`s in the chain must

claim to be able to implement it. The registry searches down the chain, asking each `Constructor` whether it `match()`es (can implement) the given `Sort` or `Operator`; see Section 2.3.5. When it finds the right `Constructor`, the registry asks the constructor to construct the appropriate `SortImpl` or `OpImpl`. The registry returns the `Impl`, and from then on everything proceeds as in Section 2.3.1. Figure 2-5 shows the process of generating code for an operator on a compound sort.

2.3.3 Curried Parameters

In most cases, there is a one-to-one mapping between IOA operators and the Java methods that implement them. Sometimes, however, it is impractical to write a method for each operator. In this case, it is useful to implement a *family* of IOA operators with the same Java method. For instance, `Char` has nullary operators for each character: `'a'`, `'b'`, etc. There are many such operators, and they differ only in the character value that they return. It is therefore convenient to implement all such operators with a single method, `CharSort.lit(`*`char`*`)`. The parameter to this method is a *`char`* that differs depending on which IOA operator is being applied at a given time. For instance, when the IOA program references the nullary operator `'a'`, the code generator outputs `CharSort.lit('a')`.

Here, the character literal `'a'` is a curried parameter; it is not present in the IOA operator application, but it is added to the Java method invocation. The operator is nullary, but the implementation method is unary. The `'a'` is built into the subclass of `Operator` object that implements `'a'`. This class is called `ExtOperator`[4]; and it supports an arbitrary number of curried parameters, which it prepends to the argument list during emission. Operators with curried parameters may be registered using `Installer.addCurriedOp()` instead of `addOp()`. Curried operators on compound sorts are not supported at this time.

[4]The implementation is due to Toh Ne Win.

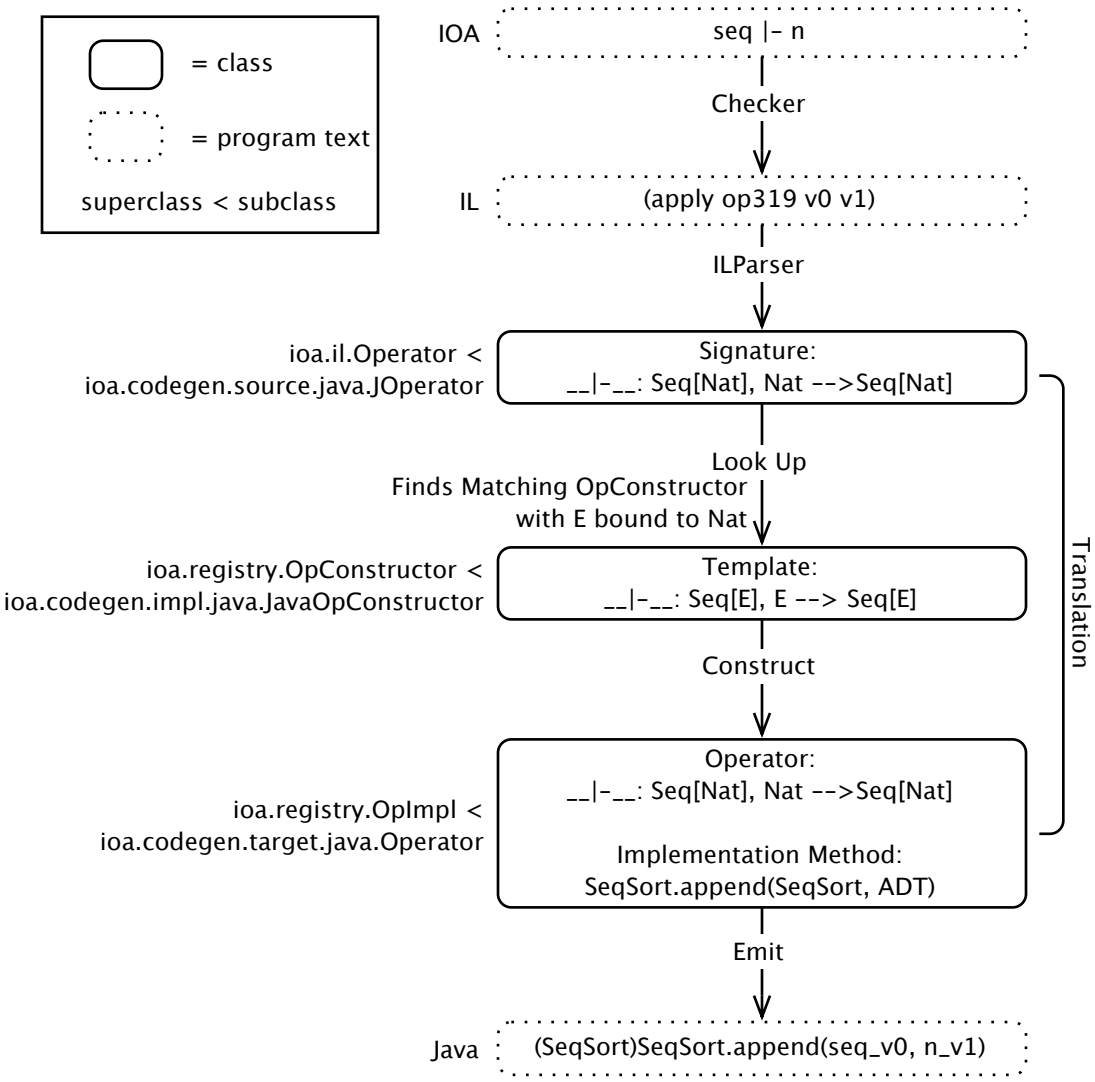

Figure 2-5: Translating and emitting an operator on a compound sort. First, the IOA program is parsed by the checker. The checker outputs an intermediate language version of the program. The `ILParser` parses this into nodes of the source syntax tree. The code generator begins the translation. When it encounters an operator on Seq[Nat], it looks it up in the registry and finds an `OpConstructor` for Seq[E], which can handle Seq[Nat]. The `OpConstructor` constructs an `Operator` node in the target syntax tree. The code generator translates the IOA operator invocation into a Java method call, and emits the text of that call with the actual parameters.

2.3.4 Looking up Dynamic Sorts and Operators

In addition to simple sorts and parameterized sorts, there is a third class of sorts called *shorthand sorts*, which includes **tuple**s, **union**s, and **enumeration**s. Shorthand sorts are dynamic in the sense that their operators are not known until compile time, when the code generator examines the user-defined data types that are introduced in the program text. Since new shorthand data types may be introduced in each IOA program, the implementation classes for these sorts must have methods that can each implement a group of operators.

For example, a tuple Tup may be defined to have two fields, a and b:

```
type Tup = tuple of a : Nat, b : Nat
```

The runtime class for tuples keeps a mapping of field names to field values. One method implements field lookup for all tuples, and another implements field setting for all tuples. These operators are implemented using curried parameters, as described in Section 2.3.3.

In the Char example from Section 2.3.3, all the possible characters are known when the *implementation class* is compiled; for shorthand sorts, the operators are not known until the *IOA program* is compiled. In the case of tuples, the code generator cannot know *a priori* what the field names are, or even how many fields there are.

The implementations of non-dynamic sorts are installed when the code generator starts up. For sorts implemented dynamically, the implementation classes are installed by the `DynamicImplRegistry`[5] the first time that their implementations are looked up. In fact, the actual registry used by the code generator is a `DynamicImplRegistry`. When looking up an implementation, it first tries delegating to its superclass, `ConstrImplRegistry`. If the normal registry cannot find the implementation, the dynamic registry tries to find one itself.

When the code generator starts up, all the registration classes for dynamic sorts, *dynamic registration classes*, notify the `DynamicImplRegistry` of their existence. Then, when the `DynamicImplRegistry` encounters a `Sort` for which the normal registry could

[5]The implementation of the dynamic registry is due to Toh Ne Win.

not find an implementation, it calls each dynamic registration class's `isDynamic(Sort)` method to ask if it can install mappings for that sort and its operators. `isDynamic(Sort)` returns *`true`* only if the dynamic registration class can implement the specified sort.

When the `DynamicImplRegistry` encounters an `Operator` that the normal registry could not find, it first tries to look up the sorts in the `Operator`'s domain. It assumes that one of these sorts introduces the operator. When the registry finds the dynamic registration class for the sort that does, it calls the dynamic registration class's `installDynamic()` method, which adds the mapping for the operator to the registry. The operator can then be looked up and returned.

A `Sort` or `Operator` is only looked for in the `DynamicImplRegistry` once. After that, it will have been added to the normal registry so there will be no need to consult the `DynamicImplRegistry`.

For example, suppose that the IOA program makes use of a **union**. The first time one of its operators, say a selection operator, is used it will not be found in the registry. The domain of the operator is a single value of the **union** type. The `DynamicImplRegistry` finds a dynamic registration class that can implement the **union**. It then asks the registration class to install all the operators introduced by that **union** type. Thereafter, operator lookups for that **union** in the `ConstrImplRegistry` will succeed.

2.3.5 Matching Compound Sorts and Operators

With compound sorts and operators, the table key is a shallow name such as `Map` or `__|-__` (as opposed to `Map[E]` or `__|-__: Seq[E], E -> Seq[E]` that tells the registry which chain of constructors to search through. Each constructor has a *template*, which indicates the pattern of sorts or operators that it can implement. The syntax for templates is defined by the grammar in Figure 2-6.

Each `SortConstructor` has a template of the form `(name subsorts)`. For instance, the `SortConstructor` for `MapSort` is `(Map 0 1)`. Since the sort variables `0` and `1` can each match any sort, this template says that `MapSort` can implement any kind of `Map`

```
sort          ::= name | "(" name subsorts ")" | sortVariable
name          ::= id
subsorts      ::= sort+
sortVariable  ::= integer

op            ::= "(" name domains range ")"
domains       ::= "(" sugaredSort+ ")"
range         ::= sugaredSort
sugaredSort   ::= sort | "%me"
```

Figure 2-6: Sort Template Grammar

that has two, possibly different, subsorts. This template would match Map[Int, Real], and it would also match Map[Int, Seq[Real]]. If the template had been `(Map 0 0)`, then the two subsorts would have to be the same; the `SortConstructor` could then match Map[Int, Int] or Map[Seq[Int], Seq[Int]], but not Map[Int, Real].

`OpConstructor`s have templates of the form `op`, and matching operators to templates works much the same as matching sorts to templates. The one difference is that operator templates may include a `%me` token. `%me` is shorthand that stands for the sort template of the sort that is currently being installed. This makes templates easier to read and write. For instance, the template for the prepend operator on sequences may be written `(__|-__ (%me 0) %me)` instead of `(__|-__ ((Seq 0) 0) (Seq 0))`.

In general, matching is a recursive process. A sort template matches a sort if it has the same shallow name and its `subsorts` can match the sort's subsorts. An operator template matches an operator if it has the same name and if the sorts in its domain and range match the pattern in the template. The recursion must bottom out because the template is of finite length and therefore has finitely many nestings. Mutual recursion is not possible because templates do not contain references, only values.

Presently, the Java code generator uses only a fraction of the flexibility provided by constructors and templates. For instance, all sequences are implemented using `SeqSort`, but the registry classes could support different implementation classes for different operators.

2.3.6 Installer

Motivation

The registry provides four methods for adding sort and operator mappings, one for each table:

- `add(String key, boolean isLiteral, SortImpl impl)`
- `add(String key, OpImpl impl)`
- `add(String name, boolean isLiteral, SortConstructor sortCon)`
- `add(String name, OpConstructor opCon)`

Unfortunately, it is cumbersome to use these methods. One must call `ConstrImplRegistry` methods to make operator and sort keys using the registry's static `makeOpKey()` and `makeSortKey()` methods. One must write classes that extend `SortImpl`, `OpImpl`, `SortConstructor`, and `OpConstructor` (which are all abstract) and create instances of them. This can certainly be done, but it needlessly complicates the interface to the registry. Therefore, a *facade* [8] class called `ioa.registry.Installer` mediates between the ADT writer and the registry, providing a clean interface that does not expose to the ADT writer any of the classes that make up the code generator.

Standard Usage

One creates an instance of `Installer` using the `ImplFactory` and passing it a reference to the registry and the name of the implementation class of that sort. The factory will then return an `Installer` specialized for code generation or simulation. Once an `Installer` has been created, *its* methods may be used to add mappings to the registry. The commonly used methods are:

- `addSort(String template)`
- `addOp(String opTemplateString, String methodName)`

And there are also some less commonly used methods to handle special cases:

- `addLiteralSort(String template)`
- `addAssignOp(String template, String methodName, String assignMethodName)`
- `addCurriedOp(String template, String methodName, Vector builtIns, boolean arrayMode)` (See Section 2.3.3.)
- `addShortcutOp(String template, String methodName, String shortcutStyle)` (See Section 2.3.8.)

The parameters for these methods are all built-in Java types for which the Java compiler can recognize literals. Thus, calls to `Installer`'s methods are concise and the programmer using `Installer` is isolated from the inner workings of the code generator. At this point in the evolution of the design, registration classes are almost entirely data.

Exceptional Usage

In most cases the LSL name of the operator should be passed to `Installer`; however, there are some exceptions due to the way the front end and intermediate language work:

- When registering a selection operator with LSL form `__.foo`, `@<sel>foo` should be passed.
- The conditional operator (which is built-into LSL) should be registered as `@<if>`.
- Multiple argument slots between the two parts of a mixfix operator should be collapsed to one. For instance, the LSL operator `[__, __, __]` should be registered as `[__]`. Though there is only one argument slot in the name, the registry will get the correct number of arguments from the template.

Implementation

`Installer` has a simple implementation: it creates the keys and `Impl` objects needed to call the registry methods listed at the top of this section. A new instance of `Installer` is created for each data type that will be added to the registry. The instance remembers the registry in which to install, the sort being installed, and the implementation class for it. `Installer`'s methods use this information, in addition to their parameters, to create instances of `ioa.codegen.target.Class` (for simple sorts), `ioa.codegen.target.Operator` (for operators on simple sorts), `ioa.codegen.impl.java.JavaSortConstructor` (for sort constructors), and `ioa.codegen.impl.java.JavaOpConstructor` (for operators on sort constructors). It then installs these instances into the registry. `Class`, `Operator`, `JavaSortConstructor`, and `JavaOpConstructor` are the canonical Java implementations of the `Impl` and `Constructor` objects described in Sections 2.3.1 and 2.3.2. Aside from being specialized for emitting Java code, they are augmented with functionality for testing (see Section 2.5.3).

`Installer` is designed to be subclassed. At present, there is one subclass for the code generator and one for the simulator. Each `Installer` is customized to install particular types of implementations. The public methods of `Installer` are *template methods* [8] and in most cases should not be overridden in subclasses. Instead, for each public method there is a corresponding "hook" method that encapsulates the functionality that is likely to differ between `Installer`s. These should be overridden to return the proper types of `SortImpl`s and `OpImpl`s. For example, the public method for installing an operator is `addOp()`. The code generator's installer overrides `opConstructorHook()` to return a `JavaOpConstructor`. The simulator's installer overrides it to return a `SimOpConstructor`.

2.3.7 Looking up Return Types

As described in Section 2.3.1, when an `Operator` emits itself as a method call on an implementation class, it also casts the return value to the proper implementation

class. Some implementation class methods always return the same type; for instance, the cardinality of a set is always represented by an `IntSort`. Others, however, return different types in different circumstances; for instance, when indexing into an array, the type of the return value depends on how the array is parameterized.

The needed information is available in the source (IOA/IL) syntax tree: each `ioa.il.Operator` stores an `ioa.il.Sort` object that represents its range. Therefore, the registry classes must find this `Sort`, look it up in the registry to find its implementation class, and give the implementation class to the `ioa.codegen.target.java.Operator`, so that it can emit the proper casts.

Ideally, perhaps, the constructor for `Operator` could include a parameter for the implementation class of its return type; that way, every `Operator` would always know how to cast its return value. However, this is not always possible. The registry is populated first by sort A and its operators, then by sort B and its operators, etc. Thus, each operator on A is constructed before sort B is even in the registry—so clearly there is no way to look up the implementation class for B while creating an operator on A. In fact, there can even be cyclic dependencies. For instance, Int and Nat define conversion operators that each depend on the other sort: nat() operates on Ints and returns a Nat, and int() operates on Nats and returns an Int.

Therefore, `Operator`'s constructor does *not* take an implementation class as a parameter. Instead, it must find it after the registry has been fully populated. The code generator has no mechanism at this time for notifying objects that the registry has been fully populated and emission has begun. Therefore, for simplicity, we use the first emission of the `Operator` as a proxy for this notification. In other words, once an `emit()` method has been called, we know that all the non-dynamic registration classes have installed their sorts and operators into the registry.

For modularity reasons, `Operator` does not know about the objects in the source syntax tree or about the registry. Therefore, to look up the implementation class of its return type, it uses a helper object called a `ReturnClassThunk`. As its name implies, `ReturnClassThunk` is a thunk that knows how to find the implementation class of the return type of an operator. Each time an `Operator` is created, it is

passed a `ReturnClassThunk`, which encapsulates access to the source syntax tree and the registry. When the `Operator` needs to find the return class, it calls the thunk's `lookupReturnClass()` method, which does the actual work.

2.3.8 Shortcutting

The implementations of the __∧__, __∨__, and if__then__else__ operators shortcut. That is, if the first clause of a conjunction is false, the second is not evaluated; if the first clause of a disjunct is true, the second is not evaluated; and only the **then** or **else** clause of a conditional is evaluated, depending on the value of the predicate. These behaviors are useful in code generation because they allow the user to guard against runtime exceptions (see Section 2.2.8).

The above operators cannot be implemented in the previously described way because Java always evaluates all method parameters. Instead, they are implemented using `ShortcutOperator`[6], which emits special forms in the target language. For instance, the if__then__else__ operator is implemented using Java's ternary operator:

IOA: a := **if** (x > y) **then** a **else** b

Java: `a_v2 =`

```
(((BoolSort)IntSort.gt(x_v0, y_v1))).booleanValue() ? a_v2 : b_v3;
```

and the __∧__ operator is implemented using Java's shortcutting `&&` operator:

IOA: c := b ∧ a

Java: `c_v2 = BoolSort.lit(b_v1.booleanValue() && a_v0.booleanValue());`

Some glue code is needed to convert between implementation classes and Java's primitive types: `BoolSort.lit()` makes a `BoolSort` from a *`boolean`*, and `BoolSort.booleanValue()` makes a *`boolean`* from a `BoolSort`.

2.3.9 Locating ADTs at Compile-Time

When the code generator starts up, it calls upon the `ADTLoader` to find registration classes that should be installed in the registry. The `ADTLoader` searches directories

[6]The implementation is due to Toh Ne Win.

and jar files for classes that implement the `Registrable` interface. Search path and exclusions are specified in the `.ioarc` file and may be overridden on the command line. This lets one, for example, choose from alternate implementations of a data type at compile-time. For more information, see [19].

2.4 Registration Classes

Each implementation class in `ioa.runtime.adt` (see Section 2.2) has a corresponding registration class in `ioa.registry.java`. By custom, the implementation class and the registration class have the same name. The job of the registration class is to populate the registry with mappings for the sort and its operators. As such, the registration class depends on the registry and is strongly coupled with its implementation class. Any changes to the specification of an implementation class must be propagated to the corresponding registration class.

Each registration class implements the `ioa.registry.Registrable` interface:

```
public interface Registrable
{
    public void install(ConstrImplRegistry reg)
        throws RegistryException;
}
```

The code generator calls the `install()` method to populate the registry with mappings from the registration class. In addition, by convention, each registration class includes two constants:

```
// name of the IOA sort
public final static String sortName  = "Foo";

// name of the implementation class
public final static String className = "FooSort";
```

Registration classes may use these when they need to refer to other sorts.

2.4.1 Standard Registration Classes

The body of the `install()` method first creates an instance of `Installer` (see Section 2.3.6) by passing it `reg` (which was a parameter of `install()`) and the two constants defined in Section 2.4. It then uses the `Installer` to add mappings for the sort and all its operators. When installing an operator, the registration class passes to the `Installer` method the operator's template (see Section 2.3.5) and the name of the implementation class method that implements it. See Appendixes A.1.3 and A.2.3 for example `install()` methods.

Note that the registration class need not install the equality, inequality, and conditional operators; these are the same for all ADTs and so they are registered automatically by the call to `addSort()` or `addLiteralSort()`.

Exception: Literal Operators

In addition to normal operators, for sorts with literals like Int and Real the registration class installs a special literal operator. The syntax for registering it looks like this:

```
installer.addOp("(@<const>lit␣()␣%me)", "lit")
```

This tells the code generator that it can use `IntSort.lit()`, to create `IntSort`s from integer literals in the source program. In order for this to work, the registration class should install the sort using `addLiteralSort()` rather than `addSort()`. The signature of the literal operator's implementation method is defined by the code generator to take a single *`int`* parameter. This is because the argument to the literal operator in the target syntax tree is a `Numeral`, a class that emits itself as an *`int`*.

2.4.2 Dynamic Registration Classes

When information about a dynamic sort (see Section 2.3.4) is known, the main registry asks the dynamic registry to actually install the dynamic sort. At this point, the dynamic registry calls upon the dynamic registration class (which extends `DynamicRegistrable`) to run its `installDynamic()` method.

`installDynamic()` uses `Installer`'s `addCurriedOp()` method to add the dynamic sort's operators to the registry. `addCurriedOp()`'s signature is like that of `addOp()`, but it has two additional arguments.

The first additional argument is a `Vector` of extra information, `builtIns`, that contains curried parameters that will be passed to the code generation method when it is called. For example, in the `lookupField()` method of `TupleSort`, the built-in argument is a `Vector` with one argument, the field name. Hence, the `__.a` operator on a tuple would be emitted as `TupleSort.lookupField("a", aTuple)` and `__.b` would be emitted as `TupleSort.lookupField("b", aTuple)`.

The second additional argument is a boolean that enables "array mode," in which all the arguments to the code generation method are passed as one array. This allows operators with a variable number of arguments. For example, `TupleSort.make()` implements the mixfix `[__ ,..., __]` operator for creating tuples. `TupleSort.make()` has to take an array of arguments because different tuples have different numbers of fields.

2.4.3 Non-Standard Registration Classes

A few non-standard operators do not fit the patterns that `Installer` knows about. The non-standard parts must be registered "manually" (as described in the first part of Section 2.3.6). They must use the `ConstrImplRegistry` instance passed to the registration class to add mappings directly, and they must define their own code emitters (that handle casting if applicable). An example in which a non-standard registration class is required is `LSeq` (Section 7.3).

2.4.4 Flexibility Without Registration Classes

Registration classes are the mechanism we have chosen to support alternate implementations of data types. For instance, one could write a wrapper for Java's `BitSet` class and use it to provide an optimized implementation of `Array[Nat, Bool]`. Ensuring that the corresponding registration class is loaded allows the code generator to find implementations for all kinds of arrays, and the load order determines which

implementation has the first chance to claim that it implements a sort.

This flexibility is possible because whenever the code generator encounters an operator it looks up its implementation in the registry. A weakness of our current scheme, however, is that operator references in implementation classes do not indirect through the registry. The operator that tells whether an element is a member of a set returns a Bool. Its implementation method assumes that Bool is implemented by `BoolSort` and calls `BoolSort.lit()` to generate the return value. Likewise, the set cardinality operator assumes that Int is implemented by `IntSort`.

A strong case can be made that Bool only needs one implementation, however for numeric sorts we desire more flexibility. The standard implementations of Int, Nat, and Real rely on Java's primitive types. They provide good performance, but are subject to the same precision limitations as the underlying Java types. The Fibonacci example from Dean's thesis overflows Java's integer bounds after a small number of steps, and certainly many real-world situations would as well. Likewise, Java's floating point arithmetic means that our implementation of Real does not conform to basic arithmetic identities. Multiplying a number by its reciprocal may not yield exactly 1, for instance.

We considered changing the implementations of the built-in numeric types to use the "arbitrary" precision numeric classes in Java's `java.math` package. However, performance tests done by Nigam [19] showed that this was not a good solution for the common case where extra precision is not needed.

In Section 2.2.3 I described how operators are officially implemented by static methods but that usually these delegate to instance methods that do the real work. This level of indirection provides a clean solution to the problem of providing alternate implementations for sorts that are accessed through hard-coded references.

We rely on the *factory method* pattern defined by Gamma et. al. [8]. Consider the case of `IntSort`, which provides the standard low-precision implementation of Int. Every instance of `IntSort` (that is created outside the `IntSort` class) is created by the factory method `lit()`. Normally, this returns an `IntSort`; but if the user has requested the high-precision Int implementation (by setting a flag in `IntSort`), it can

return a `BigIntSort`, which uses `BigInteger` as its underlying representation instead of *`int`*.

`BigIntSort` is a subclass of `IntSort`, and it overrides all the methods that create `IntSort`s. Other classes are free to call the static `IntSort` methods directly; since these delegate to instance methods, Java's dispatch will ensure that `BigIntSort`s are properly treated as such.

2.5 Test Classes

2.5.1 Testing Implementation Classes

The correctness of the generated code depends on the correctness of the implementation classes. Therefore, each implementation class has a corresponding *test class*, implemented using the JUnit testing framework [2]. See Appendix A.2.4 for an example. By convention, the test class for `FooSort` is called `FooSortTest`. Test classes live in the `ioa.test.junit.runtime.adt` package and must extend `TestCase`, which is provided by JUnit. They follow the standard JUnit pattern, containing:

- A `main()` that calls `junit.textui.TestRunner.run(suite())`. This lets one test a particular class by invoking `java` on its test class.
- A `setUp()` method that creates variables (typically instances of ADTs) that will be shared by the test methods. In JUnit terminology, the `setUp()` method creates the *fixture*.
- A `suite()` method whose body is *`return new`* `TestSuite(FooSortTest.`*`class`*`);` (where `FooSort` is the name of the ADT being tested). This tells JUnit to create a suite of all the tests for the ADT.
- Test methods for `hashCode()`, `equals()`, and each implementation method. The name of a test method *must* begin with `test`. The rest of the test method name should be the name of the method being tested. (For instance, `BoolSortTest`

contains a `testLte()` method that tests `BoolSort`'s `lte()` method.) Test methods consist of a series of calls to `assert()` and `assertEquals()`, which are provided by JUnit. Together, the assertions perform black box tests, which test based on the cases in the specification, and glass box tests, which test based on the cases in the implementation.

In addition, each test class must be listed in the `AllADTsTest` class. This can be accomplished by adding a new line to `AllADTsTest`'s `suite()` method that says
`result.addTest(FooSortTest.suite());`

Once this has been done, one can test the new ADT along with all the old ones by running `java ioa.test.junit.runtime.adt.AllADTsTest`, or by running `make adt-impl` in the toolkit's `Test` directory. To test a single ADT, run `java ioa.test.junit.runtime.adt.FooSortTest`.

If an equality assertion fails, JUnit will print out the value it expected and the value it received. The values may then be compared to track down the bug. To take advantage of this feature, ADTs should override `Object.toString()` to "unparse" themselves. By convention, simple ADTs unparse themselves into IOA syntax. Container ADTs unparse themselves into s-expressions, which are more compact than the IOA syntax necessary to construct them.

2.5.2 Testing Registration Classes

It is also important to test the registration class to make sure that the signatures of the operators that it registers match the signatures of the operators that the front-end outputs. This is accomplished by creating an IOA program that uses all of the sort's operators (see Appendix A.2.5) and running it through the code generator. These test programs are stored in the toolkit's `Test` directory. Instructions for creating and running the tests are displayed by the `make help` and `make add-help` commands in the directory.

2.5.3 Catching Bugs in the Implementation/Registration Interface

The tests in Section 2.5.2 do not ensure that the registration class maps operators to the correct methods, or even to methods that exist. Manual checking must be used to determine the correctness of the mapping, but the `CorrespondenceTest` class can increase confidence in the registration class by checking that the methods it registers actually exist in the implementation class.

`CorrespondenceTest` uses the list of `Registrables` found by the `ADTLocator` to decide which classes to test. It works by using a *mock object* [17] that poses as a registry to each registration class. When the registration class installs an operator, `CorrespondenceTest` checks that the implementation class includes a suitable method to implement it. In order to do this, it uses the `getMethodName()` and `getNumParameters()` methods of `Operator` or `JavaOpConstructor` to get information about the method being registered. These two methods are included in the `JavaMethod` interface, so `CorrespondenceTest` will work with any registered object that implements `JavaMethod`.

One problem with this design is that since `Operator` implements `JavaMethod`, so will any class that extends it (to provide non-standard functionality, for instance); and therefore `CorrespondenceTest` will think that it can test the subclass. In fact, the subclass may be so non-standard (see Section 7.3.3) that the information provided by the `JavaMethod` accessors no longer makes sense. To handle this situation, `JavaMethod` includes another method, `isTestable()`, that lets test classes determine whether the registered object claims it can be tested as described above. Subclasses of `Operator` that do non-standard things should override `isTestable()` to return *`false`*, thus preventing spurious errors when `CorrespondenceTest` is run.

2.6 Recipe for Writing ADTs

To add a new ADT to the code generator, you must:

- Create an implementation class in the `ioa.runtime.adt` package.
  - Make sure that it extends `ioa.runtime.adt.ADT` (Section 2.2.1).
  - Include a public static method for each operator (Section 2.2.3).
  - Be sure to override the non-static `equals()`, `hashCode()`, and `toString()` methods from `java.lang.Object` (Section 2.2.2).
  - Create a static `construct(Parameterization)` method (Section 2.2.5).
  - Make the class `Serializable` (Section 6.1).
  - Add methods for converting to and from s-expressions (Section 7.1).
- Create a registration class in the `ioa.registry.java` package (Section 2.4). Use `Installer` to install the sort and its operators (Section 2.3.6).
- Write a JUnit-based test class for the implementation class and add it to `AllADTsTest` (Section 2.5.1). Be sure to test `equals()` and `hashCode()`.
- Write an IOA test for the registration class and hook it up to the Makefile-based tester (Section 2.5.2).

To add a new dynamic sort, there are only a few differences:

- Some operators are likely to use curried parameters and are registered using `addCurriedOp()` (Section 2.3.3).
- The `install()` method should simply add a stub instance to `DynamicImplRegistry`:

```
DynamicImplRegistry.addDynReg(new TupleSort());
```

- The registration class should have a *`boolean`* `isDynamic(Sort` *`sort`*`)` method that returns *`true`* if the given sort can be implemented dynamically by the registration class. `DynamicImplRegistry` will query this method whenever a dynamic sort is in need of installation.

- The registration class should have an `installDynamic(Sort sort)` method that does the actual installing. Install the necessary operators by calling `addCurriedOp()` and/or `addOp()` here. `DynamicImplRegistry` will call this method once `isDynamic()` returns true. Remember to call `addSort()` to install the regular comparison and conditional operators.

2.7 Sharing ADTs with the Simulator

The abstract data types and infrastructure described in this chapter were originally developed for the IOA code generator. Concurrent with the code generator work, Chefter, Ramírez-Robredo, and Dean [4, 20, 5, 12] developed a simulator that interprets IOA. The two projects faced similar issues in matching IOA sorts and operators with Java implementations. In fact, the simulator had a registry and its own Java implementations of some basic IOA data types[7].

For obvious reasons, we determined that the code generator and simulator should use the same ADT implementations at runtime. They now use the same implementation classes, the same registration classes, and the same registry. The difference is in the mappings they install into the registry. In each case, the registration class gets an `Installer` from a factory and uses it to populate the registry. For the code generator, the factory returns a `CGInstaller`; for the simulator it returns a `SimInstaller`.

Table 2.1 shows the correspondence between the classes in the code generator and those in the simulator. Mappings to the classes on the left are installed by `CGInstaller` and mappings to the ones on the right are installed by `SimInstaller`.

In the code generator's target syntax, `Operator` nodes have an `emit()` method that prints out Java code to call the static class method that implements the operator. In the simulator's syntax tree, `BasicOpImpl` nodes have an `apply()` method that is supposed to perform the work of the operator. To do this, they use Java's reflection

[7]The registry worked in a similar manner. In fact, the code generator's (and later the shared) registry was based on earlier simulator work [20] by Ramírez-Robredo. The main difference is that in the old design the ADT implementations were interwoven with the code that registered them, which made adding new ADTs difficult. More detailed comparison between the two is contained in [25]

Code Generator	Simulator
ioa.codegen.target.java.Class	ioa.simulator.impl.BasicSortImpl
ioa.codegen.impl.java.JavaSortConstructor	ioa.simulator.impl.SimSortConstructor
ioa.codegen.target.java.Operator	ioa.simulator.impl.BasicOpImpl
ioa.codegen.target.java.ShortcutOperator	ioa.simulator.impl.ShortcutOpImpl
ioa.codegen.target.java.ExtOperator	ioa.simulator.impl.ExtOpImpl
ioa.codegen.impl.java.JavaOpConstructor	ioa.simulator.impl.SimOpConstructor
ioa.codegen.impl.java.JavaShortcutOpConstructor	ioa.simulator.impl.SimShortcutOpConstructor

Table 2.1: Class Correspondence Between the Code Generator and Simulator

API to look up the implementation method and invoke it. Curried parameters are handled in the same way as in the code generator, and shortcutting operators are similarly special-cased.

2.8 ADT Library

2.8.1 Standard IOA ADTs

This section gives an overview of the implementation classes for the data types that are built into the IOA language.

BoolSort

Bool is a predefined sort in LSL and IOA, so *every other sort depends on its implementation*, `BoolSort`. `BoolSort`s are interned: there is only one object for *`true`* and one for *`false`*. Other ADT implementations may wish to access these values; for instance, the set membership operator always returns an instance of `BoolSort`. Three public class methods[8] are provided for doing this: `True()`; `False()`; and `lit()`, which converts a Java boolean into a `BoolSort`.

Authors: Joshua A. Tauber, Michael J. Tsai

[8] `True()` and `False()` are capitalized (contrary to our naming convention) to avoid conflicts with the Java keywords *`true`* and *`false`*.

IntSort

`IntSort` is a straightforward implementation of integers using Java's `int`. *Other ADTs that return integers depend on IntSort.* For instance, the `count` operator implemented by `MsetSort` returns `IntSort`s that indicate the multiplicities of elements in the set. To support uses such as this, `IntSort` provides a public class method `lit()`, which converts Java `Integer`s into `IntSort`s. The `nat()` conversion operator is implemented, but the front-end currently does not know how to parse it.

The sort has an alternate implementation, `BigIntSort`, that uses Java's `BigInteger` to handle larger integers[9] (for instance, those generated by `Fibonacci`). `IntSort` maintains a flag (which may be set when the runtime starts up) for switching between these two implementations. This is further described in Section 2.4.4.

Authors: Joshua A. Tauber, Michael J. Tsai, Atish Dev Nigam

ArraySort

`ArraySort` maps indices to values. The space of possible indices is very large—in theory, it is unbounded. Therefore, `ArraySort` uses a sparse representation based on `Hashtable`. Each array has a constant that specifies the value at indices for which no value was specified (thus allowing arrays to be total). Two-dimensional `Array`s are implemented in the same class, as `Hashtable`s mapping pairs of indices to values. When an `ArraySort` is created, it can optionally have a constant value, which acts as a default for indices that are not explicitly mapped in that `Hashtable`.

In IOA, each element of an array is considered a separate state variable, and the array notation `array[i] := j` is merely a shorthand for modifying them (not the array itself). During code generation, statements of that form are desugared to `array = assign(array, i, j)`. The desugaring mechanism is general; any operator application appearing on the left hand side of an assignment will be desugared so that the new left hand side is the first operand and the new right hand side is an operator application with the original operands plus the original right hand side. Operators that can

[9] `BigInteger` supports integers of up to 2^{32} decimal digits.

appear on the left hand side of an assignment are registered with `Installer` as assignment operators. This involves providing an additional parameter `assignMethodName`, which is the name of the method that implements the operator when it is used as an lvalue.

Author: Michael J. Tsai

CharSort

`CharSort` is a simple wrapper for Java *char*s. It implements lexicographic comparison by comparing Unicode values, which means that `'A'` is less than `'a'`. `CharSort`s are not interned at present, although this would be a simple space and time optimization.

The front-end does not (yet) specially handle character literals, as it does for integers and reals. Instead, each character literal is a nullary operator whose range is a `Char`. These operators are all implemented by the `CharSort.lit()` method and registered with curried parameters, as described in Section 2.3.3.

Author: Michael J. Tsai

MapSort

`MapSort` is much like `ArraySort`. It differs in that IOA `Map`s do not support constant values, and it is implemented with a `HashMap` from the Java Collections framework. (This is the new way of doing things in JDK 1.3. It should be faster than the `Hashtable` that `ArraySort` uses, which was all that was available when it was written with JDK 1.1.) Objects in the domain and range must be immutable, and objects in the domain must properly override `equals()` and `hashCode()`.

Author: Michael J. Tsai

MsetSort

`MsetSort` implements IOA multisets using a `HashMap` to map elements (`Object`s) to their multiplicities (`Integer`s). Objects stored in the set must be immutable and must properly override `equals()` and `hashCode()`.

Author: Michael J. Tsai

NatSort

`NatSort` is implemented similarly to `IntSort`. It throws an exception if asked to contain a negative number. The int() conversion operator is implemented but the front-end currently does not know how to parse it. Like `IntSort`, `NatSort` has an alternate implementation (`BigNatSort`) that uses `BigInteger`s.

Authors: Michael J. Tsai, Atish Dev Nigam

RealSort

`RealSort` implements real numbers using Java *`double`*s. At present, it ignores floating point precision issues. As a result, values that should be equal may be reported as unequal (and vice-versa). However, we also provide `BigRealSort`, an alternate implementation that uses `java.math.BigDecimal`. This provides better, but not perfect precision, and it still cannot represent irrational numbers such as π.

Authors: Michael J. Tsai, Atish Dev Nigam

SeqSort

`SeqSort` is a `Vector`-based implementation of IOA sequences. Sequences are homogenous lists of values that are indexable by integers.

Authors: Joshua A. Tauber, Michael J. Tsai

SetSort

`SetSort` implements IOA sets (as well as ChoiceSet) using a `HashSet` for speed. Because it was written before `MsetSort`, it is not based on `MsetSort`. Also, it is probably faster this way.

Authors: Michael J. Tsai, Toh Ne Win

StringSort

`StringSort` implements IOA's String data type using Java `String`s. An IOA String is the same as Seq[Char], a sequence of characters.

Author: Michael J. Tsai

EnumSort, TupleSort, and UnionSort

Because information about user-defined types is not known until after the program has been parsed, these sorts have dynamic registration classes to create their operator implementations on-demand. See Section 2.3.4.

Authors: Toh Ne Win, Laura G. Dean, Michael J. Tsai

2.8.2 Other ADTs

The implementation classes described in this section are part of the standard IOA distribution, but they are not built into the language. Non-built-in data types are defined in external LSL files, which are referenced using the `-path` switch of `ioaCheck`.

LSeqSort

See the description in Section 7.3.

Author: Michael J. Tsai

PQSort

`PQSort` implements priority queues using a binary heap. Its elements must be `ComparableADT`s. `PQSort` supports both queues where the largest element is at the head and queues where the smallest element is. These correspond to the IOA types PQ[E] and PMQ[E], respectively. The registration classes for both of these sorts reference the same `PQSort` implementation class. The registration classes install the same operator implementations, except that `add` is implemented by `addMax()` or `addMin()` depending on the type of queue. Initially, an empty `PQSort` does not know which style of queue it is implementing. However, by the first `add` it is able to set that information based on which implementation method was called.

Authors: Atish Dev Nigam, Michael J. Tsai

NullSort

`NullSort` implements Null, which is a lifted type. It can hold either one value of some specified subsort or the distinguished value nil.

Authors: Toh Ne Win, Michael J. Tsai

StackSort

`StackSort` implements stacks using a `Vector`.

Authors: Atish Dev Nigam, Michael J. Tsai

TimedInvocationSort

`TimedInvocationSort` implements TimedInvocation, which is a tuple of an invocation, a time, and a node. Ordinarily, this could be implemented as a tuple without the need for adding a new ADT. However, TimedInvocations differ from other tuples in that they are totally ordered and support an ordering operator. Thus, `TimedInvocationSort` is a `ComparableADT`. `TimedInvocationSort` *does not* obey the standard contract that `compareTo()` must return `0` exactly when `equals()` returns *`true`*. This is because the ordering of TimedInvocations is specified only in terms of their nodes and times. There should never be two TimedInvocations with equal nodes and times, but different invocations, because the order would be undefined. If this case ever arises, `compareTo()` will throw a `RepException`.

Author: Michael J. Tsai

TreeSort

`TreeSort` implements binary trees. Each node stores a data object and may have 0, 1, or 2 children.

Authors: Toh Ne Win, Michael J. Tsai

Chapter 3

The Interface Generator

The algorithm automaton communicates with the external environment (e.g., users at consoles or input and output files) through an auxiliary interface automaton. The composition of these two automata and the auxiliary network automata is a primitive automaton. After we remove the nondeterminism from the primitive automaton, we generate Java code for it. The interface automaton is defined algorithmically in terms of the the algorithm automaton. Producing the interface automaton, given the algorithm automaton, is the job of the *interface generator*.

The interface automaton interacts with its local environment by sending and receiving invocations. An *invocation* is simply the label of an action from the algorithm automaton combined with its actual parameters. The user can send input into the system by passing the interface automaton an invocation of an input action. Likewise, the external environment observes the system through the invocations of output actions that it receives from the interface automaton.

At a high level, composing the algorithm and interface automata has the effect of making all of the algorithm automaton's input and output actions internal. The only non-internal actions of the composite automaton are:

```
input  getInvocation(inv:IOA_Invocation)
output putInvocation(inv:IOA_Invocation)
```

These are both introduced by the interface automaton, and we require that the

algorithm automaton not have actions with these labels.

3.1 Type Definitions

Invocations consist of an action label and actual parameters, both of which depend on the definition of the algorithm automaton. Therefore, the representation of an invocation cannot be defined in advance; it must be generated from the algorithm automaton.

Among the first things the interface generator does is parse the algorithm automaton and create a new sort, IOA_Invocation, to represent invocations on it[1]. IOA_Invocation is a shorthand sort defined as:

```
type IOA_Invocation = tuple of action:IOA_Action,
                              params:Seq[IOA_Parameter]
```

This definition is fixed, but since IOA_Action and IOA_Parameter differ with the algorithm automaton, IOA_Invocation must also.

The action field of the tuple identifies the action to be invoked. The IOA_Action sort is defined as an enumeration of the names of the actions in the algorithm automaton. It does not take into account transition cases because after composition there will only be one transition with each label.

The params field of the tuple is a sequence of the actual parameters. The parameters of an action may have different sorts, for instance Int and Bool, but the elements of an IOA sequence must always have the same sort. Therefore, we define IOA_Parameter as a union of all the possible sorts that an actual parameter can have. Through this indirection, the params sequence can then hold heterogenous actual parameters[2].

The IOA_Parameter sort is defined as a union of all the (unique) sorts of the actual parameters of actions in the algorithm automaton. The tags of the union are the

[1]The prototype interface generator avoided the complexity of introducing new sorts and operators by using integers instead of invocations. Each transition had a number, and the parameters were restricted to integers.

[2]An alternate design would have been to create a specialized invocation **tuple** for each kind of action, however this would have made the resulting IOA (and the Java to generate it) longer and more complex.

```
sort          ::= name | name "'_" subsorts "_'"
subsorts      ::= sort ("'" sort)*
```

Figure 3-1: Grammar for representing sorts and their subsorts as IOA identifiers.

names of the sorts. For instance, if p:IOA_Parameter contains an integer, its value may be accessed with p.Int.

It is also possible for the union to contain compound sorts. In this case, the name of the union tag is constructed using the grammar in Figure 3-1. For example, the tag for the Array[Int,Bool] sort is Array'_Int'Bool_'. The two different subsort delimiters, '_ and _', are necessary to handle nested compound sorts.

The interface generator contains a general mechanism for introducing new shorthand sorts and their operators. For instance, if the programmer wants to introduce a new **union** he can call a single method, passing it the name of the sort and the names and sorts of the fields. The symbol table will then be populated with the **union** sort, its accompanying tag **enumeration**, and all the operators on these two sorts. I hope that this mechanism can be reused in the development of other intermediate language transformation tools.

3.2 States of the Interface Automaton

The interface automaton has two state variables:

```
states
  stdin: LSeq[IOA_Invocation],
  stdout: LSeq[IOA_Invocation]
```

The two sequences implement FIFO invocation buffers. head(stdin) is the oldest input invocation (the one that appears earliest in the trace), and head(stdout) is the oldest output invocation.

The LSeq sort is identical to the standard Seq sort except that it supports locking. LSeq is defined by the following LSL trait:

```
LockableSequence(E): trait
  includes
    Sequence(LSeq for Seq)
  introduces
    lockStdin, lockStdout: LSeq[E] → LSeq[E]
    unlockStdin, unlockStdout: LSeq[E] → LSeq[E]
```

The semantics of the new operators are not specified in this LSL trait. The operators' only purpose is to allow the interface generator to mark regions where the generated Java code needs to obtain exclusive access to shared variables. This is described in more detail in Section 7.3.

3.3 Input Actions of the Interface Automaton

For each output action[3] a of the algorithm automaton, the interface automaton contains an input action a' with the same number and types of parameters. a performs the "work" of the action, while a' translates an invocation of a into an IOA_Invocation. The composite automaton contains an action that combines the effects of a and a'. The action is hidden during the composition.

The effect of a' is:

```
stdout := lockStdout(stdout);
stdout := stdout ⊢ inv;
stdout := unlockStdout(stdout)
```

The first and last statements establish and relinquish exclusive access to the output invocation buffer. The middle statement appends inv to the buffer, where inv is an IOA_Invocation whose action is the name of a and whose parameters are the parameters of a.

For instance, if a is:

```
output foo(i: Int, b: Bool)
```

then the effect of a' is:

[3]The output action named SEND is ignored because it deals with network output rather than console output. It is a special stub that will be replaced by MPI calls.

```
stdout := lockStdout(stdout);
stdout := stdout ⊢ [foo, {} ⊢ Int(i) ⊢ Bool(b)];
stdout := unlockStdout(stdout)
```

The second line of this requires some explanation. The right hand side of the assignment creates a new sequence by appending an invocation to the old sequence. The [__, __] operator for making the invocation takes two operands. foo is a nullary operator that produces an IOA_Action. {} ⊢ Int(i) ⊢ Bool(b) creates a sequence by appending two elements to the empty sequence. The Int operator takes an integer i and packages it in an IOA_Parameter. The Bool operator does the same thing with a boolean.

3.4 Output Actions of the Interface Automaton

For each input action[4] a of the algorithm automaton, the interface automaton contains an output action a' with the same number and types of parameters. a performs the "work" of the action (as defined by the algorithm automaton), while a' translates an IOA_Invocation into an invocation of a.

The precondition of a' is that the action and parameters in the first invocation in stdin match the current action and parameters. For instance, if a is:

```
input foo(i: Int, b: Bool)
```

then the precondition of a' is:

```
pre head(stdin).action = foo ∧
    len(head(stdin).params) = 2 ∧
    tag(head(stdin).params[0]) = Int ∧
    head(stdin).params[0].Int = i ∧
    tag(head(stdin).params[1]) = Bool ∧
    head(stdin).params[1].Bool = b
```

The effect of a' is to remove the invocation from stdin. It looks like:

[4]The input action named RECEIVE is ignored because it deals with network input rather than console input. It is a special stub that will be replaced by MPI calls.

```
eff stdin := lockStdin(stdin);
    stdin := tail(stdin);
    stdin := unlockStdin(stdin);
```

3.5 getInvocation and putInvocation

Conceptually, the interface automaton contains the actions:

```
input  getInvocation(inv:IOA_Invocation)
output putInvocation(inv:IOA_Invocation)
```

getInvocation receives an invocation from the environment and appends it to stdin. putInvocation removes an invocation from stdout and sends it to the environment. In the current version of the code generator, the environment is a collection of input and output files, one per computation node.

The interface generator does not, however, output the definitions of these actions. Since they do not depend on the algorithm automaton, they are implemented once and for all in the code generator's runtime library. The class `ioa.runtime.io.Stdin` implements getInvocation, and the class `ioa.runtime.io.Stdout` implements putInvocation. Neither of these actions needs to be scheduled, and each of them runs in its own Java thread. This is how we model the fact that input can arrive at any time.

The runtime handling of invocations is described in more depth in Chapter 7.

3.6 Example

This section shows the interface automaton that is generated for an automaton that implements a bank. The banking automaton is a modified version of one presented by Toh Ne Win and Gustavo Santos [32]. Its workings are not important for the purposes of this example; the main point is to see the form of the interface automaton.

3.6.1 Algorithm Automaton **Banking**

The algorithm automaton has output actions OK and reportBalance and input actions requestDeposit, requestWithdrawal, and requestBalance.

```
% Implementation of null possibility
type OpRec = tuple of  loc: Int,
                       seqno: Int,
                       amount : Int,
                       reported: Bool
type BalRec = tuple of loc: Int,
                       value: Int

% Defines sums over a set
uses NonDet
uses ChoiceSet (OpRec)
uses ChoiceSet (BalRec)

automaton Banking
signature
  input
    requestDeposit(n: Int, i: Int) where n > 0,
    requestWithdrawal(n: Int, i: Int) where n > 0,
    requestBalance(i: Int)
  output
    OK(i: Int, x : OpRec),
    reportBalance(n: Int, i: Int)
  internal
    doBalance(i: Int,
              tempChosenOps : Set[OpRec],
              amount : Int)
  states
    ops: Set[OpRec] := {},
    pending_ops : Set[OpRec] := {},
    reported_ops : Set[OpRec] := {},
    pending_bals : Set[BalRec] := {},
    done_bals : Set[BalRec] := {},
    bals : Set[BalRec] := {},
    lastSeqno: Array[Int, Int] := constant(0),
    chosenOps: Set[OpRec] := {}

  transitions
    input requestDeposit(n, i)
    eff
      lastSeqno[i] := lastSeqno[i] + 1;
      ops := insert([i, lastSeqno[i], n, false], ops);
      pending_ops := insert ([i, lastSeqno[i], n, false],
                             pending_ops)

    input requestWithdrawal(n, i)
    eff
      lastSeqno[i] := lastSeqno[i] + 1;
```

```
    ops := insert([i, lastSeqno[i], -n, false], ops);
    pending_ops := insert ([i, lastSeqno[i], -n, false],
                           pending_ops)

  input requestBalance(i)
  eff
    pending_bals := insert ([i, 0], pending_bals);
    bals := pending_bals ∪ done_bals

  output OK(i, x)
    % X isn't a real parameter, it just helps avoid choose
  pre
    x ∈ ops ∧ x.loc = i ∧ ¬x.reported
  eff
    ops := insert(set_reported(x, true), delete(x, ops));
    pending_ops := delete (x, pending_ops);
    reported_ops := insert(set_reported(x, true),
                           reported_ops)

  output reportBalance(n, i)
  pre
    [i, n] ∈ done_bals
  eff
    done_bals := delete([i, n], done_bals);
    bals := pending_bals ∪ done_bals

  internal doBalance(i, tempChosenOps, amount)
    pre
      [i, 0] ∈ pending_bals
    eff
      chosenOps := tempChosenOps;
      pending_bals := delete([i, 0], pending_bals);
      done_bals := insert([i, amount], done_bals);
      bals := pending_bals ∪ done_bals
```

3.6.2 Interface Automaton BankingInterface

This section shows the actual output of the interface generator. The interface automaton has states stdin and stdout. For each input action in the algorithm automaton it has an output action, and vice-versa. The input actions remove IOA_Invocations from stdin, and the output actions add IOA_Invocations to stdout.

The output has been converted from the intermediate language back to IOA using the IL-to-IOA translator developed by Reimers [21] (and subsequently enhanced by Nigam and myself). I have hand-modified the output slightly by adding linebreaks and indentation to format it for this page. Note that the output is not quite valid

IOA. The IL-to-IOA translator needs to be improved so that type definitions are outputted in an order that does not have any forward references.

```
type IOA_Invocation = tuple of action: IOA_Action,
                               params: Seq[IOA_Parameter]
type IOA_Parameter = union of Int: Int,
                              OpRec: OpRec,
                        Set'_OpRec_': Set[OpRec]
type IOA_Action = enumeration of OK, reportBalance
type BalRec = tuple of loc: Int, value: Int
type OpRec = tuple of loc: Int,
                      seqno: Int,
                     amount: Int,
                   reported: Bool
automaton BankingInterface
  signature
    input
     OK(i: Int, x: OpRec),
     reportBalance(n: Int, i: Int)
    output
     requestDeposit(n: Int, i: Int) where n > 0,
     requestWithdrawal(n: Int, i: Int) where n > 0,
     requestBalance(i: Int)
  states
    stdin: LSeq[IOA_Invocation] := {},
    stdout: LSeq[IOA_Invocation] := {}
  transitions
    output requestDeposit(n: Int, i: Int) where n > 0 case 1
      pre ((((((head(stdin).action) = (requestDeposit)) ∧
        ((len(head(stdin).params)) = 2)) ∧
        ((tag(head(stdin).params[0])) = (Int))) ∧
        ((head(stdin).params[0].Int) = n)) ∧
        ((tag(head(stdin).params[1])) = (Int))) ∧
        ((head(stdin).params[1].Int) = i)
      eff stdin := lockStdin(stdin);
          stdin := tail(stdin);
          stdin := unlockStdin(stdin)
    output requestWithdrawal(n: Int, i: Int) where n > 0 case 1
      pre ((((((head(stdin).action) = (requestWithdrawal)) ∧
        ((len(head(stdin).params)) = 2)) ∧
        ((tag(head(stdin).params[0])) = (Int))) ∧
        ((head(stdin).params[0].Int) = n)) ∧
        ((tag(head(stdin).params[1])) = (Int))) ∧
        ((head(stdin).params[1].Int) = i)
      eff stdin := lockStdin(stdin);
          stdin := tail(stdin);
          stdin := unlockStdin(stdin)
    output requestBalance(i: Int) case 1
      pre ((((head(stdin).action) = (requestBalance)) ∧
        ((len(head(stdin).params)) = 1)) ∧
        ((tag(head(stdin).params[0])) = (Int))) ∧
        ((head(stdin).params[0].Int) = i)
```

```
  eff stdin := lockStdin(stdin);
      stdin := tail(stdin);
      stdin := unlockStdin(stdin)
input OK(i: Int, x: OpRec) case 1
  eff stdout := lockStdout(stdout);
      stdout := stdout ⊢ ([OK, (({}) ⊢ (Int(i))) ⊢
                                  (OpRec(x))]);
      stdout := unlockStdout(stdout)
input reportBalance(n: Int, i: Int) case 1
  eff stdout := lockStdout(stdout);
      stdout := stdout ⊢ ([reportBalance,
                             (({}) ⊢ (Int(n))) ⊢
                                 (Int(i))]);
      stdout := unlockStdout(stdout)
```

Chapter 4

The Next Action Determinator

The output of composing the node automaton with the interface automaton is a nondeterministic automaton: more than one action may be enabled at a time and the effects of the actions may be nondeterministic. The next action determinator (NAD) is a source-to-source transformer. It takes as input the nondeterministic automaton and outputs a modified version, the *NAD automaton*, that is *next-action deterministic*. That is, at any given time only one locally controlled[1] action of the NAD automaton is enabled.

The user can specify a schedule for the next-action deterministic automaton. The schedule selects actions to execute and chooses parameters for them. The user also replaces the remaining explicit nondeterminism (in the form of **choose** expressions) with deterministic values. The final scheduled automaton is called *next-state deterministic*. With all the nondeterminism removed, it is then possible to generate runnable code to implement the automaton in an imperative programming language.

This chapter will give an overview of the NAD transformation from a syntactic point of view (in terms of the IOA language). Vaziri et. al. [29] describe the precise semantic transformation (in terms of the I/O Automaton model) and show that it conforms to the syntactic one. Note that the form of the NAD automaton described in that paper is slightly different from the one generated by the NAD tool in the IOA

[1] A *locally controlled* action is one that is either **output** or **internal**. **input** actions are, by definition, always enabled.

toolkit. In particular, the NAD implementation uses integers for transition identifiers instead of enumerations[2].

4.1 Kinds of Nondeterminism

IOA programs contain both explicit and implicit nondeterminism. Explicit nondeterminism includes **choose** statements, which represent nondeterministic values, possibly constrained by a **where** clause. Implicit nondeterminism includes the label of the next action and the values of its parameters.

The goal of the NAD transformation is to make all implicit nondeterminism explicit, i.e. transform all implicit nondeterminism into **choose** statements. The user can then determine the **choose**s by replacing them with deterministic values.

4.2 The NAD Transformation

The NAD takes as input a primitive automaton A. It requires that the automaton not have a state variable named PC or an action named Scheduler. The output of the NAD is a new automaton $\text{NAD}(A) = A'$ that contains no implicit nondeterminism. Further, $\textsc{Traces}(A) = \textsc{Traces}(A')$, i.e. the transformation preserves the external behavior of A. If the name of N is Name, the name of A' is NameNAD. A' differs from A as described in the following subsections.

4.2.1 States of A'

A' contains all the state variables of A.

A' has an integer state variable PC, which represents a program counter. Each transition of A' is numbered with a unique integer, a *transition index*. PC takes on only those values. PC is initialized to s, where s is the index of the Scheduler transition.

[2]The NAD tool was implemented before the paper was written; and at the time, the toolkit's intermediate language did not support shorthand sorts such as enumerations.

A' contains a new state variable for each free variable in the actual parameters of its locally controlled transitions. These new states have the same types and intermediate language identifiers as the the parameters they were derived from. This means that everywhere A referenced one of the actual parameters, A' will reference the state variable derived from it. The names of the state variables are the same as the names of the parameters, except that a prefix is added to make them unique. The prefix is the letter "t" followed by the transition index and an underscore.

4.2.2 The Scheduler Transition

A' contains a new internal transition called Scheduler. It has transition index s and is defined as follows:

Scheduler has no parameters. The precondition of Scheduler is PC $= s$. The effect of Scheduler does the following:

- For each state variable v added above, the effect contains a statement of the form $v :=$ **choose**. This makes assignment of the parameter values explicitly nondeterministic.

- The effect contains the statement PC := **choose** p **where** p $\neq s$. This makes the choice of the next transition explicitly nondeterministic. The **where** clause ensures that the next transition will not be Scheduler[3].

- The final part of the effect is a conditional statement. Let i_i be the indices of the locally controlled non-Scheduler transitions and let p_i be their preconditions. Then the conditional statement is:

```
if ¬((PC = i_1 ∧ p_1) ∨
     (PC = i_2 ∧ p_2) ∨
     ...
     (PC = i_n ∧ p_n)) then
         PC := s
fi
```

[3]At present, it does not ensure that the choice is a legal transition index, however.

If the PC has been assigned the index of an enabled transition, the conditional does nothing and that transition will be the next to run. Otherwise, the conditional sets PC to Scheduler's transition index so that a new transition and parameters can be picked.

4.2.3 Transitions of A'

For each locally controlled transition t of A, A' contains a transition t'. t' is similar to t with a few changes:

- In the precondition and effect, all references to the parameters of t are replaced by references to the corresponding new state variables in A'.
- The precondition has a new conjunct requiring that PC equals the transition index of t'. This ensures that t' only runs when it has been scheduled and that only one transition is enabled at a time. Thus, A' will be next-action deterministic.
- For every actual parameter in t, t' contains an actual parameter that is an identical term except that the free variables have been replaced by fresh variables.
- The precondition of t adds conjuncts requiring that these fresh variables are equal to their corresponding new state variables.
- At the end of the effect, PC is assigned the value s, to return control to Scheduler.

4.3 Summary

The Scheduler action decides which transition will run next. Its precondition is that the program counter is equal to its index. Its effect is to assign **choose** values to each of the stand-in state variables (Section 4.2.1) that represents one of the chosen action's parameters. This is equivalent to assigning parameter values for each of the transitions, then choosing which transition to run (by setting the program counter

accordingly). It is possible that the precondition of the chosen transition will not be satisfied. In this case, Scheduler sets the program counter to its own index. The next time it runs it will hope for better luck.

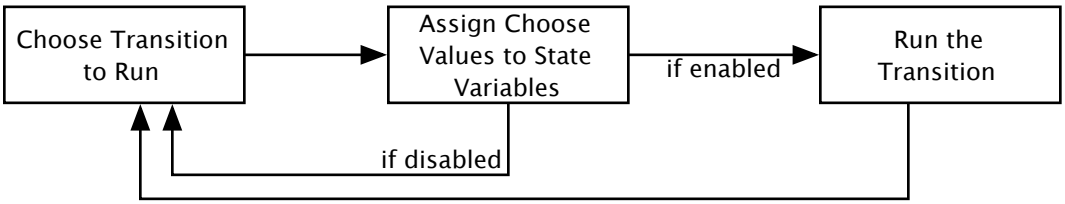

Figure 4-1: The Scheduler Transition's Main Control Loop

Normally, transitions can be executed at any time. However, in the scheduled automaton, Scheduler decides which transition should run when. Thus, each locally controlled transition is modified so that it runs only when given permission by Scheduler, and so that it relinquishes control to Scheduler after it has finished running. To do this, a conjunct is added to each transition's precondition requiring that the program counter is equal to its transition index. Additional precondition conjuncts require that the actual parameters of the transition match the corresponding state variables created above. Subsequent references to these actual parameters are replaced by references to the state variables. Finally, at the end of the effect clause, the transition sets the program counter to Scheduler's transition index.

4.4 Example

This section shows a simple example of an automaton before and after the NAD transformation.

4.4.1 Nondeterministic Automaton Adder

The Adder automaton has a single input action add whose effect is to compute a sum and store it in a state variable. Once it has done this, the result action can output the sum.

```
automaton Adder
  signature
    input add(i, j:Int)
    output result(k:Int)
  states
    value: Int,
    ready: Bool := false
  transitions
    input add(i, j)
      eff
         value := i + j;
         ready := true
    output result(k)
      pre
         k = value ∧ ready
      eff
         ready := false
```

4.4.2 Next-Action Deterministic Automaton AdderNAD

The AdderNAD automaton has two additional state variables, one for the program counter and one for the parameter to the lone output action. The output action's precondition has been modified to require that the program counter is equal to its index and that the parameter is equal to the stand-in state variable. Its effect has been modified to set the program counter to Scheduler's index. Finally, the new Scheduler transition chooses values for the program counter and the output action's parameter. If the output action is not enabled, it sets the program counter to its own index so that it will run again.

```
automaton AdderNAD
  signature
    input add(i, j:Int)
    output result(k:Int) % transition index 1
    internal Scheduler   % transition index 0
  states
    value: Int,
    ready: Bool := false,
    PC: Int := 0,
    t1_k: Int
  transitions
    input add(i, j)
      eff
         value := i + j;
         ready := true
    output result(k)
```

```
    pre
      PC = 1 ∧ t1_k = k ∧ t1_k = value ∧ ready
    eff
      ready := false;
      PC := 0
  internal Scheduler
    pre
      PC = 0
    eff
      t1_k = choose
      PC = choose
      if ¬(PC = 1 ∧ k = value ∧ ready) then
        PC := 0
      fi
```

4.5 Evaluation of the NAD

The next action determinator is easy to use, but adding the scheduling information is hard. The chief problem is that the user thinks in terms of the node automaton, but he must schedule the NAD automaton. This is difficult because IOA for the NAD automaton is generated by the IL-to-IOA translator, which does not preserve the organization, formatting, or comments from the original algorithm automaton source file. Also, the NAD automaton contains the original automaton's parameters renamed as state variables.

Second, there is the minor problem that the schedule (i.e. the resolutions of the **choose**s) needs to be pasted into skeleton Scheduler transition in the NAD output each time the node automaton changes. This is actually more annoying than it first appears. Chances are that deciding which transition to run will require some temporary variables for use in the scheduling logic; these must also be added to the NAD automaton after each regeneration.

Third, and more serious, is that the schedule must be written in very low-level terms. For instance, to "fire" a transition the user must locate and set the NAD-generated state variables that represent the transition's parameters. Then he must set the PC state variable to the program counter value that corresponds to the transition. The values of the program counter and the names of the state variables both depend on the transition indices. However, the transition indices depend on the in-

termediate language representation of the IOA program[4]. As such, they are unstable; semantics-preserving changes to the node automaton, such as changing the order of the transitions in the source file, can change the transition indices. There is no warning to the programmer when the numbers have changed; his schedule will simply stop working or behave unexpectedly.

One solution to this latter problem would be to use better transition identifiers. As described in [29], the PC variable could be an **enumeration** of transition names instead of integers. This would solve the stability problem and allow the IOA checker to report errors when the schedule contained invalid transition identifiers (because the name would not be in the definition of the enumerated type). However, the state variable names would still depend on the transition names. Also, there is still the problem of ensuring that the parameter state variables are set properly; the checker will not signal an error if they are uninitialized.

[4]At present the `BasicIntILFactory` assigns increasing integers to each transition it creates.

Chapter 5

Compiling the Nondeterminism Resolution Language

The implementation of the NAD was completed in August 1999. At the time, we were far from having a working system for compiling IOA, and an easier to use NAD was not on the critical path. I revisited the scheduling problems in the winter of 2002. By this time, Antonio Ramírez-Robredo had completed his thesis [20] on a new IOA simulator. One component of Ramírez-Robredo's work was a set of extensions to the IOA language for resolving **choose** nondeterminism and scheduling transitions. These NDR (for "nondeterminism resolution") language constructs appear as annotations in IOA programs. Most tools ignore the annotations, but the simulator uses them. In fact, the mechanism for scheduling transitions to select valid executions forms the backbone of the simulator's interpreter.

NDR solves most of the problems with the NAD in Section 4.5. (The user still has to schedule interface automaton actions that he did not write.) Since the annotations are added directly to the IOA source file, there is no need to copy and paste each time the automaton is changed. The semantic checker can verify that each fired transition actually exists, and also that the number and types of its actual parameters are suitable.

Thus, rather than trying to reinvent these features in the NAD, I decided to add NDR support to the code generator. NDR fragments could be compiled directly to

Java, and the NDR schedule could form the main[1] method of the generated Java class. An added benefit is that NDR **schedule**s and **choose** resolutions can be shared between the simulator and the code generator. The user can specify this information once and use it with both tools at different points in the development process. Finally, NDR has the benefit of not modifying the underlying IOA code[2].

The remainder of this section explains the relevent NDR syntax elements and shows how they may be translated to Java.

5.1 NDR Language Extensions

Ramírez-Robredo's thesis [20] introduces IOA syntax extensions to support simulation and paired simulation. The extensions include labeling invariants, labeling transition definitions, resolving nondeterminism by scheduling transitions and determining **choose** values, and specifying proofs of simulation relations. The extensions were then modified somewhat by Dean [5] as she reimplemented them in a newer version of the IOA front-end.

This section is concerned with the extensions for resolving nondeterminism. I have extracted the relevent BNF productions from Dean's thesis and will explain them below.

```
basicAutomaton ::=   "signature" formatActions+ states
                                  transitions tasks?
```

Figure 5-1: IOA Grammar for an Automaton

Automaton definitions have been extended to allow an optional **schedule** block that specifies transitions to be scheduled. The **schedule** can reference (but not modify) the state variables of the automaton and evaluate arbitrary terms to decide which

[1] "Main" as in the method that oversees everything, not as in the actual "main" method.

[2] However, tools such as the composer must be aware of NDR so that they can preserve the annotations.

```
basicAutomaton  ::=   "signature" formatActions+ states
                                  transitions tasks? schedule?
schedule        ::=   "schedule" states? "do" NDRProgram "od"
NDRProgram      ::=   NDRStatement;*
NDRStatement    ::=   assignment
                    | NDRConditional
                    | NDRWhile
                    | NDRFire
NDRConditional  ::=   "if" predicate "then" NDRProgram
                           ("elseif" predicate "then" NDRProgram)*
                           ("else" NDRProgram)? "fi"
NDRWhile        ::=   "while" predicate "do" NDRProgram "od"
NDRFire         ::=   "fire" actionType actionName
                             actionActuals? transCase?
                    | "fire"
```

Figure 5-2: Annotated IOA Grammar for an Automaton

transitions to schedule. The **fire** statement schedules the running of a transition and specifies its actual parameters. Execution begins at the top of the schedule block. When a **fire** statement is reached, the simulator executes the specified transition and then returns to the statement immediately following the **fire** in the schedule.

```
choice          ::=   "choose" (variable "where" predicate)?
```

Figure 5-3: IOA Grammar for a Choice

IOA **choose** statements have been extended to allow optional determinator blocks that specify values that may be chosen. Determinators are like schedules except that instead of **fire**ing transitions the determinator **yield**s values.

Determinators "remember" where they left off; execution of a determinator resumes after the last-executed **yield** statement (rather than at the top of the block). Therefore, to specify that a sequence of values should be yielded it is sufficient to include a sequence of **yield** statements. The first time the determinator is executed it will yield the value from the first statement, the second time from the second

```
choice          ::=   "choose" (variable ("where" predicate)?)?
                              choiceNDR?
choiceNDR       ::=   "det" "do" NDRProgramY "od"
                    | NDRYield
NDRProgramY     ::=   NDRStatementY;*
NDRStatementY   ::=   assignment
                    | NDRConditionalY
                    | NDRWhileY
                    | NDRYield
NDRConditionalY ::=   "if" predicate "then" NDRProgramY
                          ("elseif" predicate "then" NDRProgramY)*
                          ("else" NDRProgramY)? "fi"
NDRWhileY       ::=   "while" predicate "do" NDRProgramY "od"
NDRYield        ::=   "yield" term
```

Figure 5-4: Annotated IOA Grammar for a Choice

statement, and so on.

The resumptive nature of these semantics was inspired by CLU iterators [15]. It makes the IOA programmer's job particularly easy because much of the control flow is handled by the language, although it complicates the job of the code generator because Java does not have native support for this type of control flow.

Unlike schedules, the contents of a determinator are surrounded by an implicit infinite loop.

5.2 Adding NDR to the Code Generator

5.2.1 Parsing NDR Constructs

The first step in adding NDR support to the code generator was parsing the NDR language extensions in the intermediate language. The simulator already had code to do this, thanks to work by Ramírez-Robredo [20]. He designed the intermediate language parser to allow elements to have *extensions*. Syntactically, an extension is an additional s-expression at the end of a list that represents an IL element. If the IL

parser detects an extension, it first uses the current factory to create the normal node for the IL element and then gives the factory the node and the extension so that the factory can add any extension-specific information to the node. The NDR language extensions are implemented as extensions by the simulator; and, as such, the logic for parsing them was embedded in the simulator's specialized IL factory. Likewise, the Java parse tree nodes for these elements were part of the simulator's IL nodes.

I separated the NDR node representations from the simulator, creating generic IL nodes for determined choices, scheduled automata, **while** loops, **fire** statements, and **yield** statements. Now the simulator nodes extend these NDR nodes to add the simulator-specific behavior (in particular, the machinery for interpreting the nodes at runtime). The logic for parsing the NDR nodes is now contained in the `NDRILFactory`. Note that I have extracted from the simulator only those of Ramírez-Robredo's language extensions that pertain to resolving nondeterminism. Other extensions, such as those for verifying simulation relations, remain part of the simulator.

5.2.2 Compiling NDR Constructs

The next step was to modify the code generator to handle these new IL nodes. My intent in implementing the NDR forms described above was to make the generated code have the same observable behavior as when the simulator interprets the NDR forms directly. In particular, I retain the CLU-iterator semantics for determinators even though they are not built into the target language (Java).

This has important implications for the generated code. The simulator walks an internal representation of an IOA/NDR program, and the simulator's control mechanism can remember, for each NDR program, where execution should resume. The generated Java code must somehow also do this. I considered two possible designs:

- In the first design, each NDR statement (or perhaps each basic block) is represented by an executable object. Executing the object performs the work of the NDR statement and passes control to the next executable object. In the case of **yield**, the executable object for the next NDR statement could be passed along

with the value to be yielded. Higher-level logic could then store the resume point for the NDR program.

- In the second design, the NDR program is translated to a Java implementation of machine code. Each NDR statement is assigned an address, and a program counter variable keeps track of the current execution point in the NDR program. More specifically, addresses can be represented by integers. Program counters are implemented by instance variables (one per NDR program). The Java translation of an NDR program is a `switch` statement on the `pc` variable, and the Java translation of each NDR statement is protected by a `case` guard.

These designs are quite different. The first is high-level and essentially amounts to generating Java code for an NDR interpreter. The second is low-level and essentially adds to Java goto's with first-class labels. I chose the second design for several reasons:

- The first design seemed harder to implement, although this is perhaps only due to my lack of imagination. Program logic would be distributed among many generated classes.

- The first design seemed harder to debug. Control logic would be encoded in the runtime connections between instances of these classes. To debug the control flow, one would have to read the generated code that hooked up these connections.

- The first design would lead to slower runtime execution. Each NDR statement would require a runtime object and a method call. In the second design, instead of objects there are Java statements; and instead of method calls there are `switch` statements, which may be compiled into conditional branches to an address offset by an index register.

The main reason for considering the first design was that the low-level approach seemed dirty. After all, goto was deliberately left out of Java. Nevertheless, the first design's only advantage was an aura of apparent cleanliness that quickly faded as I

tried to imagine implementing it. Explicit addresses and program counters may be ugly and tedious to interpret in one's head, but Java admits a straightforward and succint implementation.

5.2.3 Handling Transitions with Parameters

In Tauber's prototype, the code generator did not handle transitions with parameters because we expected that it would only be generating code for automata outputted by the next action determinator (see Chapter 4). The NAD changes transition parameters into state variables, thus leaving all the transitions parameterless.

Extending the code generator to support parameters was straightforward. For each action formal parameter, the code generator adds a formal parameter to the Java method that implements the transition. The type of the parameter corresponds to the implementation class for the IOA parameter's sort (see Section 2.2). The actual parameters are supplied by **fire** statements in the **schedule**; see Section 5.5.5.

One complication is that IOA transition parameters may be arbitrary terms[3]. For instance, an action may take parameters i and i+1. The code generator creates a dummy parameter for each IOA formal parameter that is not a simple variable reference.

5.3 Java Translation Overview

A nice property of the second design is that IOA terms may be compiled into the same Java code regardless of whether they are part of an NDR program or an IOA program. Statements in NDR programs, on the other hand, translate differently from statements in IOA programs.

Each statement is ultimately contained by an NDR **schedule** block or **det** block. These blocks each translate into Java methods whose bodies are `switch` statements

[3]In the output of the composer, the parameters will be in a restricted form, however it is worth handling the general case so that the code generator can compile automata that have not been generated from a composition.

inside infinite *while* loops.

Primitive statements—assignments, **yield**s, and **fire**s—translate into a single *case* and associated block. The *case* ensures that the block is only executed when the program counter has the appropriate value. The last statement of the block sets the program counter to the number of the next statement to execute.

Programs (which consist of lists of statements) translate into lists of *case* statements as shown in Figure 5-5. Although NDR programs may be nested, the Java translation removes nesting so that each top-level NDR program is a single flat list of *case*s. This impedes readability somewhat but leads to a simpler implementation.

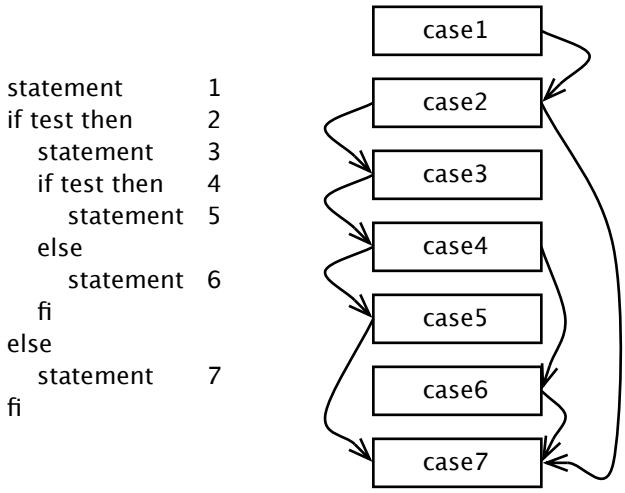

Figure 5-5: Nested statements are flattened into a sequence of statements that jump from one to the next. Arrows indicate control flow.

5.4 Java Translations of NDR Blocks

5.4.1 Schedules

A **schedule** block translates into a Java method named *schedule* and an instance variable, `schedulePC`, for its program counter. The body of the method consists of an infinite *while* loop labeled `scheduleLoop`. Inside the loop is a *switch* statement that chooses a *case* based on the value of `schedulePC`. `schedulePC` is initalized to the address of the first statement in the **schedule** transition. After executing this

statement, `schedulePC` will have been assigned a new value. Control will proceed to the end of the *`switch`* and jump back up to the top of the infinite loop. The *`switch`* will then transfer control to the *`case`* corresponding to the new value of `schedulePC`.

The last *`case`* represents the end of the IOA schedule. It breaks out of the schedule loop, allowing the automaton's *`schedule`* method to return. If the schedule is infinite, this last *`case`* will never be reached.

A **schedule** block can optionally contain declarations for state variables to be used within the schedule. These are translated into Java instance variables, just like normal IOA state variables would be.

5.4.2 Determinators

A **det** block translates into a Java method named `resolveChoose`N and a corresponding instance variable `choosePC`N. N is a unique number identifying the **det** block and is assigned based on the order of the **det**s in the program. The program counter is represented as an instance variable rather than a local variable in `resolveChoose`N because it must be remembered between executions of the **det** block. This is necessary to support the resumptive semantics for **yield**.

The enclosing **choose** translates to a method call on `resolveChoose`N, which returns the value yielded. The inside of `resolveChoose`N has the same structure as *`schedule`*. The body consists of an infinite *`while`* loop with a *`switch`* inside. One difference from *`schedule`* is that **det** blocks contain an implicit NDR **while** loop. Thus, the first *`case`* of the *`switch`* in a **det** block is the compilation of the NDR **while** loop's test. The loop itself is implemented the same way as an explicit **while** loop, described in Section 5.5.4.

5.5 Java Translations of NDR Statements

5.5.1 Programs

NDR programs (lists of statements) translate into a list of *case* statements. In order to do this, the code generator translates each statement of the program into a list of *case*s, making sure that the next address after the last *case* of each statement is the address of the first *case* of the next statement. The address of the program is the address of the first *case*, and the last *case* jumps to the address of the first statement after the program.

For instance, the program:

```
yield 2;  % PC 6
yield 3   % PC 7
```

translates to:

```
case 6: /* yield */
{
    choosePC0 = 7;
    return ((NatSort)NatSort.lit(2));
}

case 7: /* yield */
{
    choosePC0 = 3;
    return ((NatSort)NatSort.lit(3));
}
```

5.5.2 Assignments

An NDR assignment translates into a list of one Java *case*. The Java first statement of the *case* updates the value of the state (or schedule) variable being modified. The second statement sets the program counter to the address of the NDR statement after the assignment. The third statement is a *break*, which effectively jumps to this address.

For instance, the assignment:

```
baz := ¬baz
```

translates to:

```
case 27: /* assignment */
{
    baz_v10 = ((BoolSort)BoolSort.not(baz_v10));
    schedulePC = 21;
    break;
}
```

Here, 21 is the address of the statement after the assignment.

5.5.3 Conditionals

An NDR conditional translates into a list of Java *case*s. The first one evaluates the test expression of the NDR conditional and sets the program counter to the address of the appropriate program. The subsequent *case*s are the translations of the NDR programs. The last *case* of each of these programs sets the program counter to the address of the statement after the IOa conditional.

For instance, the conditional:

```
if baz then
```

translates to:

```
case 25: /* conditional test */
{
    if ( baz_v10.booleanValue() )
    {
        schedulePC = 26; break;
    }
    else
    {
        schedulePC = 21; break;
    };
}
```

Here, 26 is the address of the **then** block, and 21 is the address of the **else** block.

5.5.4 Loops

An NDR **while** loop is similar to a conditional. The first *case* evaluates the test expression. If it is true, the program counter is set to the address of the first statement in the loop. Otherwise, it is set to the address of the first statement after the loop. The last statement in the loop jumps back up to the test *case*. If it is known at compile time that the loop test will always be true, the address of the statement after the loop is set to be -42, which should signal to the reader that it is not a valid address.

For instance, the loop:

```
while true do
```

translates to:

```
case 21: /* while loop test */
{
    if ( ((BoolSort)BoolSort.True()).booleanValue() )
    {
        schedulePC = 23; break;
    }
    else
    {
        schedulePC = 22; break;
    };
}
```

Here, 23 is the address of the first statement in the loop, and 22 is the address of the first statement after the loop.

5.5.5 fire Statements

An NDR **fire** statement translates into a list of one Java *case*. The first statement of the *case* sets the program counter to the address of the NDR statement after the **fire**. Since the program counter is implemented as a Java instance variable, it will be remembered until control returns to the *schedule* method. The second statement of the *case* calls the method that implements the fired transition, passing it the actual parameters. The transition's precondition and **where** clause are verified inside the

method that implements the transition (see Section 8.1).

For instance, the **fire** statement:

```
fire internal bar(1, 2)
```

translates to:

```
case 24: /* fire */
{
    schedulePC = 25;
    action_bar(((IntSort)IntSort.lit(1)),
               ((IntSort)IntSort.lit(2)));
}
```

Here, 25 is the address of the statement after the **fire**.

5.5.6 yield Statements

An NDR **yield** statement translates into a list of one Java *case*. The first statement of the *case* sets the program counter to the address of the NDR statement after the **yield**. Again, the program counter will be remembered until the next execution of the current **choose**-resolving method. The second statement of the *case* evaluates the expression to be yielded and returns it.

For instance, the **yield** statement:

```
yield 2
```

translates to:

```
case 6: /* yield */
{
    choosePC0 = 7; return ((NatSort)NatSort.lit(2));
}
```

Here, 7 is the address of the statement after the **yield**.

5.6 Example

Appendix B shows an example automaton and its Java translation. The automaton is not meant to "do anything" other than exercise the NDR language constructs

concisely.

5.7 Comparisons With the Simulator

Like the simulator, the generated Java code does not check that the value yielded by a determinator satisfies the **choose**'s **where** clause, however this would be a straightforward addition.

The simulator allows schedules to **fire** input actions. This probably does not make sense for the code generator. After composition with the interface automaton (see Chapter 3) all the original input actions will have become internal. The remaining input action, getInvocation, is implemented specially. It runs in a separate thread handled by `ioa.runtime.io.Stdin` and is not scheduled by the programmer. Nevertheless, the code generator supports firing of input actions to allow local testing using the same IOA files as the simulator. The `ioa.codegen.source.java.JNDRFire` class can optionally disallow firing of input actions[4].

Since the code generator and simulator share ADT implementations (see Section 2.7), they also share the implementation of the NonDet pseudo-trait (see Section 2.5 of [20]). This trait provides operators for generating random numbers and booleans, which are useful in writing schedules.

The NDR syntax additions admit a **fire** statement that does not specify a transition or parameters. The simulator interprets such a statement as a directive to fire an enabled transition uniformly at random. The code generator does not support this kind of **fire** and will instead report a compile-time error.

Future work: This restriction was an oversight; it should be straightforward to generate Java code that checks which transitions are enabled and uses reflection to fire one of them.

When the simulator runs, it outputs a trace showing which transitions executed and when state variables were modified. I have extended the base `Automaton` class so

[4]Although, it should always allow firing of RECEIVE, the lone input action that will remain after composition with the auxiliary automata (Section 6.5).

that it will produce similar traces. At the beginning of each transition, the generated Java code calls the base class's `enteredTransition()` method; at the end of the transition it calls `exitedTransition()`. These methods print the trace information to the console. The base class maintains a map that records the values of all the state variables. If, at the end of a transition, it finds that the state variables have changed, it prints out a list of the changed variables and their new values.

This output is not exactly the same as the simulator's. The code generator should be modified to output step information. Also, when the simulator evaluates an assignment statement, it marks the lvalue as being modified. The generated Java code compares the *values* of the state variables to see if they have changed. Thus, the simulator considers a reassignment of the same value to a variable to be a state change, while the generated code does not.

Future work: Future versions of the code generator could take the simulator's approach, at the expense of a little more complexity and reduced performance.

Future work: An interesting project would be to modify the code generator to output AspectJ [14] instead of Java. Then the tracing and variable monitoring could be implemented using aspects, instead of having the code generator insert calls to tracing methods.

Future work: Another short project would be to extend the `Automaton` base class to support commands for limiting the number of steps run and setting the random number seed. This would allow more direct comparison of traces produced by the simulator and the generated Java code. When run for the same number of steps with the same random numbers, the output should be identical, modulo the state variable caveat mentioned above. It would also be useful for the `Automaton` base class (or the code generator shell) to have a command-line switch for controlling whether traces are printed (or generated in the first place).

Chapter 6

MPI

The code generator produces Java code that runs on a network of workstations. The Java virtual machines on the different workstations communicate via a small subset of mpiJava [1]. As described in Tauber's thesis proposal [23], we model the MPI service as an automaton that provides reliable FIFO channels. A lower-level automaton corresponds to the programmatic interface of MPI, and we introduce auxiliary automata that compose with the lower-level automaton to implement the abstract communication service.

6.1 Serializing ADTs

In order to transmit IOA values between machines, the ADT implementation classes must be converted into bytes that are transferred across the network and reconstituted as Java objects. mpiJava provides support for sending and receiving Java objects that implement Java's `Serializable` interface, and we take full advantage of this.

The `Serializable` interface is actually a marker that tells the Java runtime that a class will allow itself to be serialized. The runtime provides a default implementation of serialization (and deserialization) that is suitable for most purposes. Most of our implementation classes therefore get this for free by implementing `Serializable`.

The only complication is for classes that intern their instances. `BoolSort` maintains canonical `TRUE` and `FALSE` objects and compares them using `==`. When a `BoolSort`

```
/**
 * When a BoolSort arrives from MPI and is deserialized,
 * it may  not == any of the canonical BoolSort instances.
 * This method, automatically invoked by Java, will substitute
 * the appropriate interned value.
 */
private Object readResolve()
    throws java.io.ObjectStreamException
{
        return value ? TRUE : FALSE;
}
```

Figure 6-1: Implementation of `readResolve()` for Booleans

arrives from the network, it must be replaced by the canonical `BoolSort` instance in the current virtual machine.

This is accomplished by implementing the `readResolve()` method (Figure 6-1). A similar technique is used by `TreeSort` and `NullSort`. The Java virtual machine will automatically call `readResolve()`, if present, when it deserializes an ADT.

Although we intend for each ADT to be `Serializable`, each class must implement the interface separately, rather than implementing it in the base `ADT` class. This encourages programmer awareness to help prevent late surprises in which a class seemingly works with Java's default serialization but fails in an unexpected way because `readResolve()` was not implemented.

6.2 Setting up MPI

Interaction with mpiJava is handled by `ioa.runtime.Automaton`, the base class from which the generated Java classes inherit. The subclass's `main()` method calls `Automaton`'s `main()`, passing the name of the class as an argument. The base class uses this to set up MPI.

MPI assigns each node automaton an integer *rank*, which serves as its address for sending and receiving messages.

6.3 Running Automata

At runtime, MPI starts a Java virtual machine at each node and loads the compiled automaton class. I have written a Perl script, `jmpirun`, that simplifies the user's interaction with MPI. The user passes `jmpirun` the name of the automaton class and (optionally) the number of machines to run it on. `jmpirun` then creates a wrapper executable that the C MPI implementation (MPICH, in our case) can start on each machine. MPICH locates the proper number of workstations from the `machines.LINUX` file and runs the wrapper executable on each of them. The wrapper executable creates a Java virtual machine and uses it to run the `main()` method of the automaton class.

The `main()` method calls back to the `Automaton` base class to start up mpiJava at the local node. It then creates the input and output threads. It adds the threads to a round-robin scheduler to ensure that each one runs at regular intervals. It then starts the three threads. The input thread collects invocations from the local environment and appends them to the input sequence. The output thread removes invocations from the output sequence and sends them to the local environment (a file or the console). The main thread runs the schedule that the user has written.

6.4 MPI Experiment

I developed a simple experiment to demonstrate that ADT serialization worked and that MPICH and mpiJava were correctly installed on the TDS network. The demonstration consists of a single Java class called `ADTTest`. Each node sets up mpiJava. One node is chosen, by rank, to be the "coordinator." Each other node uses mpiJava to send a message to the coordinator. One node sends an array of integers. Another sends a union of a map, a set, and a multiset. A third node sends a tuple of primitive types. When the coordinator receives a message, it prints it out on the user's console. After the coordinator has received a message from each other node it closes down mpiJava and exits.

6.5 Future Work

The code generator currently lacks an important piece of its interface to MPI. After composition with the auxiliary network automata (two for each remote node), the automaton will have two non-hidden actions that must be implemented:

```
input SEND(m:M, const i, const j)
output RECEIVE(m:M, const i, const j)
```

Here, M is a message type and i and j are source and destination nodes. These actions may be implemented in the `Automaton` base class. The network automata each have queue state variables that will appear as tuples in the composition. The implementations of SEND and RECEIVE will need to locate the proper tuple based on i and j in order to access the queues. I suspect that this will be possible using Java's reflection interface.

Chapter 7

Invocations and Runtime I/O

This chapter describes how nodes interact with their local (non-network) environment.

7.1 ADT S-Expressions

We have already established (in Section 1.3) that each node's input and output files consist of lists of invocations, and that invocations are represented at runtime as IOA tuples (Section 3.1). But how are the tuples stored on disk? One option would be to rely on Java's serialization feature as we do for sending data types across the network. However, for testing and debugging purposes we would like to be able to read and edit the input files manually. Further, it would be nice to be able to read arbitrary output invocations even though we do not have a translator from data type instances back to the surface syntax. We need a text representation that is human readable but that can be unambiguously parsed into `TupleSort`s, ideally without any auxiliary information about the automaton. The representation should be extensible so that it can support additional data types that the user may define.

The natural choice is s-expressions. They are human readable and easily navigable in text editors such as Emacs. Even better, the IOA toolkit already includes classes for representing, parsing, and printing them. I chose to represent each ADT as a list of two or more elements. The first element is a string that is the name of the ADT's implementation class. Subsequent elements encode the value of the data type

instance.

In the next two sections, I will explain how ADTs are unparsed into s-expressions and how they are reparsed back into ADTs. Then I will show a few representative examples of how ADTs are encoded as s-expressions.

7.1.1 Converting ADTs to S-Expressions

Each ADT instance must be able to convert itself to an s-expression. Programmatically, this is enforced by having the ADT implement the `SPrintable` interface. This requires it to implement a single method `toSValue()` that returns an s-expression, which is represented by the `SValue` class. The method can encode the ADT-specific information (such as the elements in a set) in any way it chooses. The only requirement is that `toSValue()` return a two-element list whose first element is the name of the implementation class. The `ADT` base class provides a helper method, `toSValue(SValue)`. When passed the ADT-specific information, it returns the two-element list headed by the implementation class name.

For an example implementation of `toSValue()`, see the `SetSort` class in Appendix A.

7.1.2 Converting S-Expressions to ADTs

The `ADT` base class is the top-level interface for reconstituting an ADT from an s-expression. It contains a static method `construct(SValue)` that takes an s-expression and returns an `ADT`. The base class does not know how to parse all the different types of ADT s-expressions; that would be impossible since the ADT library is extensible. It therefore relies on the implementation classes to parse themselves. It reads the first element of the s-expression to get the name of the implementation class. It then uses Java's reflection interface to locate that class and ask it to parse the remainder of the s-expression.

Each implementation class includes a static `construct(SValue)` method that creates an instance given the remaining part of the s-expression. This functionality is

provided with a method rather than a constructor because I intend for it to call the ADT's constructor. This will allow it to reuse any defensive representation checks, e.g. that a natural number is not negative. It also provides easy support for interned objects (Section 6.1); `construct()` can return a shared instance rather than creating a new one.

For an example implementation of `construct(SValue)`, see the `SetSort` class in Appendix A.

7.1.3 Encoding Examples

Individual ADT implementations can decide how to represent themselves as s-expressions. A simple example is booleans. The first element of the s-expression is the name of the implementation class (as required) and the second element is either the string "true" or the string "false". Thus, a true boolean is encoded as:

```
(ioa.runtime.adt.BoolSort true)
```

When the ADT base class is asked to parse this s-expression, it will call `BoolSort`'s `construct(SValue)` method, passing it the s-expression "true". `BoolSort` will then use this to decide which canonical instance to return.

Integers are handled in a similar way, with a numeral in place of "true" in the above example. The `IntSort` class uses Java's `Integer` class to parse the numeral string into a number.

Array instances are represented as follows:

```
(ioa.runtime.adt.ArraySort ((ioa.runtime.adt.IntSort 7)
                            (((ioa.runtime.adt.IntSort 1)
                              (ioa.runtime.adt.IntSort 2))
                             ((ioa.runtime.adt.IntSort 3)
                              (ioa.runtime.adt.IntSort 4))
                             ((ioa.runtime.adt.IntSort 5)
                              (ioa.runtime.adt.IntSort 6)))))
```

The 7 is the array's constant value, the value that all unassigned indices take on. The list that follows consists of pairs of mappings: 1 maps to 2, 3 maps to 4, and 5 maps to 6.

Tuples are represented similarly to arrays, as pairs of associations. An instance of the tuple:

```
type myTuple = tuple of a: Bool, b: Int
```

where a is **true** and b is 50 is represented as:

```
(ioa.runtime.adt.TupleSort ((a (ioa.runtime.adt.BoolSort true))
                            (b (ioa.runtime.adt.IntSort 50))))
```

Enumerations encode the names of the tags in the sort and the zero-based index[1] of the current instance's tag. For instance, the d instance of the union:

```
type myEnum = enumeration of a, b, c, d, e
```

is represented as:

```
(ioa.runtime.adt.EnumSort (3.0 a b c d e))
```

Unions are simply a combination of an enumeration and an object value. For instance, the union

```
type myUnion = union of a: Bool, b: Int
```

can hold a value of type Bool or Int. A myUnion that holds the integer 6 would be written in IOA as b(6). The s-expression representation is:

```
(ioa.runtime.adt.UnionSort ((ioa.runtime.adt.IntSort 6)
                            (ioa.runtime.adt.EnumSort (1.0 a b))))
```

[1]The indices are always integers, but they appear as decimals due to the way the s-expression printer works.

7.2 Threading

At runtime, each node uses three Java threads[2].

An *input thread* receives invocations from the environment and sequences them inside the interface automaton; it implements getInvocation (see Section 3.5). An *output thread* removes invocations from the interface automaton's output sequence and passes them on to the environment; it implements putInvocation (see Section 3.5). Finally, the *main thread* executes all the other transitions and interacts with MPI. Multiple threads are necessary because the Java platform does not support non-blocking I/O.

The input thread is managed by `ioa.runtime.io.Stdin`. At present, it reads invocation s-expressions from a file that is named according to the node's rank, an integer identifier assigned by MPI that is unique among the nodes in the system (see Chapter 6). This allows us to provide input for all the nodes in a series of numbered files that are accessible through the network file system. This is convenient for testing and development purposes. `Stdin` can also read node-local input files, and it might be extended to receive interactive input from a console or GUI.

The output thread is managed by `ioa.runtime.io.Stdout`. It writes invocations out as s-expressions in a file named according to the node's rank. In the future, `Stdout` might be modified to generate output in a more human readable form.

Future work: It would be nice to have a mechanism for specifying delays in input files. One solution would be to augment the syntax of the NDR language with a delay construct that would be outputted as a special invocation. Another option would be to introduce a reserved transition name that the input thread would treat specially.

7.3 Lockable Sequences and `LSeqSort`

The stdin and stdout sequences (see Section 3.2) are implemented by static variables in the `Automaton` base class. From an IOA perspective, these sequences are simply

[2]mpiJava may allocate additional threads for its own use (see Chapter 6); in any case, the generated code and the runtime we provide use three threads directly.

LSeq[IOA_Invocation] state variables inside a composite automaton with atomic transitions.

From a Java perspective, however, the sequence objects are shared between threads. The stdin state variable is appended to by the getInvocation action and tailed by the original input actions (which are now internal). getInvocation runs in the input thread, but the other actions run in the main thread. Sharing the objects between threads introduces two requirements for correctness:

- Steps must be taken to ensure that each thread has an up-to-date reference[3] to the shared sequence object. An analogous problem arises with stdout and putInvocation.
- Additionally, the sections of code that modify these sequences must run atomically to prevent race conditions:
  - The input thread could load the current value while the main thread was updating it. Then the update from the main thread would be lost when the input thread wrote back.
  - The main thread could load the current value while the input thread was updating it. Then the update from the input thread would be lost when the main thread wrote back.

Both of these requirements are addressed using locking. The shared sequences are represented using `LSeqSort`s (see Section 7.3), which are just like `SeqSort`s except that they also have support for locking and unlocking.

Since the interface automaton had state variables for sequences stdin and stdout, the automaton class will have instance variables to represent these sequences. Transitions (other than getInvocation and putInvocation) access the sequences through the state variables, so the generated code accesses them through the instance variables.

[3]Since our ADT implementations are immutable, the potential problem is not of mutating an object at the wrong time but of retaining a reference to an obsolete object.

The input thread maintains a local variable to access the `LSeqSort` representing stdin, and the output thread has a local variable to access the `LSeqSort` representing stdout.

Crucially, the threads must agree on the values of the `LSeqSort`s. The base `Automaton` class maintains two static variables that always point to the official values of stdin and stdout. When a thread wants to access one of the sequences, it calls `Automaton.getStdin()` or `Automaton.getStdout()` to retrieve the value of `Automaton`'s static variable; it then copies the return value into its instance variable (main thread) or local variable (input or output thread). Likewise, when a thread finishes modifying one of the sequences, it writes its instance variable or local variable back to `Automaton`'s static variable using `setStdin()` and `setStdout()`.

Locking is used to ensure that a thread can atomically obtain the value of a sequence, create a new value, and update the official value.

7.3.1 Java Code to Update Stdin

When the main thread removes a value from stdin, it follows this sequence of steps (also see Figure 7-1):

0. Initially, the automaton instance's instance variable points to a possibly stale `LSeqSort` object. `Stdin`'s local variable and the `Automaton` class's static variable both point to the current `LSeqSort` ("old" in the figure).

1. The main thread acquires a lock on the `Stdin` class.

2. The automaton instance calls `Automaton.getStdin()` to update its instance variable to point to the official `LSeqSort` object that represents stdin ("old" in the figure).

3. The automaton instance computes a new `LSeqSort` object that is the tail of the previous sequence and updates its instance variable to point to this `LSeqSort` ("new" in the figure).

4. The automaton instance calls `Automaton.setStdin()` to update the official stdin with the `LSeqSort` value held by its instance variable.

5. The main thread releases the lock on the `Stdin` class.

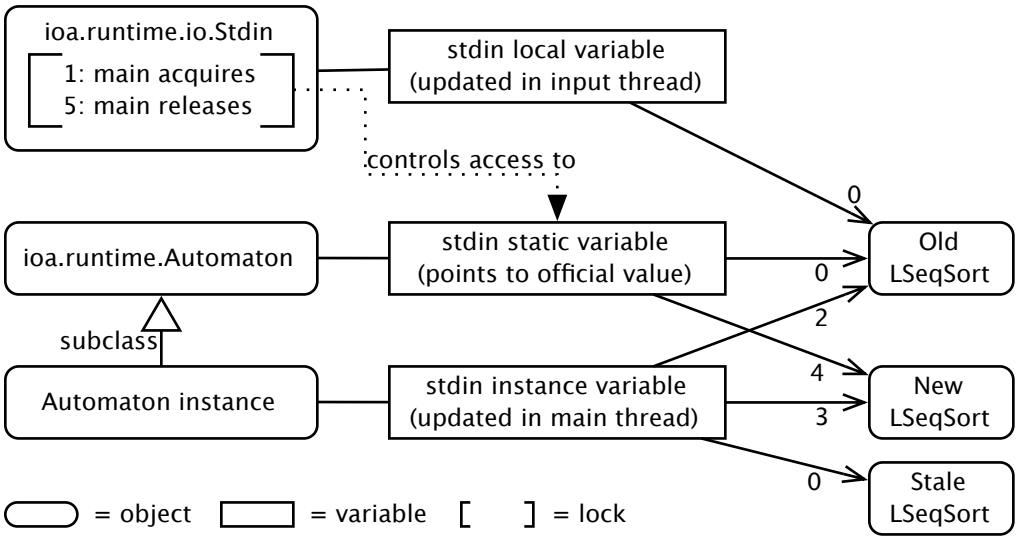

Figure 7-1: Updating Stdin from the Main Thread

The Java code generated to accomplish update stdin is shown in Figure 7-2. The numbered steps correspond to the operations in the preceding list. Some quirks of the translation will be explained in Section 7.3.2.

```
stdin_v0 = stdin_v0;              // artifact of translation
synchronized ( ioa.runtime.io.Stdin.class )       // step 1
{
    stdin_v0 = ioa.runtime.Automaton.getStdin();  // step 2
    stdin_v0 = (LSeqSort)LSeqSort.tail(stdin_v0); // step 3
    stdin_v0 = stdin_v0;          // artifact of translation
    ioa.runtime.Automaton.setStdin(stdin_v0);     // step 4
}                                                 // step 5
```

Figure 7-2: Java Code for Updating Stdin From the Main Thread

There are three other cases that employ a similar locking strategy:

- When the main thread adds a value to stdout.
- When the input thread adds a value to stdin.

- When the output thread removes a value from stdout.

In Java, each object has a lock. It does no good to obtain a lock on an `LSeqSort` object because it is immutable; each time stdin is changed, `Stdin`'s variable points to a new object. Therefore, the `Stdin` and `Stdout` classes are used as the lock objects.

7.3.2 IOA Code to Update Stdin

The Java code fragment in Figure 7-2 appears in each method of the automaton class that implements a (former) input action of the algorithm automaton. Since the rest of the action's effects must be translated, the fragment is generated by translation from IOA. In contrast, the corresponding code in the input thread is hand-written and compiled into the runtime library. Therefore, the input to the code emitter must contain a series of IOA statements that translates to the above code.

The basic idea is to add special IOA operators that translate to the *`synchronized`* block and the code for calling the methods on the `Automaton` class. These operators are added as part of the special LSeq sort, which (not surprisingly) is implemented by `LSeqSort`. LSeq (lockable sequence) is IOA's standard sequence data type extended to support four new operators: lockStdin, unlockStdin, lockStdout, and unlockStdout.

Schematically, the IOA should look something like:

```
lockStdin(stdin);
stdin := tail(stdin);
unlockStdin(stdin)
```

The middle line does the tailing and translates to Java in the normal way. The special operators in the other two lines would then translate into the locking and synchronization code.

This simple idea is complicated by the fact that lockStdin(stdin) and unlockStdin(stdin) are not valid IOA statements. They are just terms. To make the IOA valid, we can put the terms on the right hand side of assignment statements. This is a hack, but it is simpler than the alternatives.

```
stdin := lockStdin(stdin);
stdin := tail(stdin);
```

```
stdin := unlockStdin(stdin)
```

We define lockStdin and unlockStdin to be the identity function, so that the IOA program will have the desired meaning. That is, the first and third lines will not have any effect except in the Java translation.

The code emitter goes about translating the three IOA statements in the normal way. In particular, the left hand side of the first assignment will translate into the left hand side of a Java assignment. In order to produce valid Java, the translation of the lockStdin operator must complete this assignment and also begin the *synchronized* block and call `getStdin()`. Therefore, the Java translation of lockStdin(stdin) is:

```
stdin_v0;
synchronized ( ioa.runtime.io.Stdin.class )
{
    stdin_v0 = ioa.runtime.Automaton.getStdin();
```

Here, `stdin_v0` is the name of the instance variable that represents stdin. The first line of Java (`stdin_v0;`) is the right hand side of the first assignment statement in Figure 7-2. That assignment has no effect; it is merely an artifact of the decision to generate the locking code by translation.

The translation of the `unlockStdin` operator is:

```
    stdin_v0;
    ioa.runtime.Automaton.setStdin(stdin_v0);
}
```

The first line of Java (`stdin_v0;`) is the right hand side of the assignment statement that was generated as the normal translation of stdin :=. Again, the assignment has no effect and is merely an artifact of the translation. The call to `setStdin()` updates the official value of stdin. Finally, the closing brace ends the *synchronized* block, thus releasing the lock.

7.3.3 Implementing the Locking Operators

Ordinarily, IOA operator invocations translate into Java method calls (Section 2.2.3). However, since the locking operators translate into different sorts of Java fragments, they need special emitters and a non-standard registration class to install them. The

LSeq sort and its standard sequence operators (head, ⊣, etc.) are registered using Installer as in Section 2.4.1.

The locking operators are registered as shown in Figure 7-3. The LSeqSort registration class declares static inner classes that extend LockOp (a helper class that extends ioa.codegen.target.java.Operator), one for each of the four new operators. Each new kind of operator overrides emitApplication() to emit itself in a particular way. For example, LockStdinOp emits code that establishes a lock on Stdin and grabs the shared LSeqSort value that represents stdin.

```
{
  public LockStdinOp() throws CGException
  {
     super();
  }

  /**
   * Emit code that will lock Stdin. <TT>opands</TT>
   * must contain a single parameter, the LSeqSort to lock.
   *
   * i.e. if the LSeqSort is v0, then emit:
   *
   * v0;
   * synchronized ( ioa.runtime.io.Stdin.class )
   * {
   *     v0 = ioa.runtime.Automaton.getStdin();
   */
  public Emitter emitApplication(Emitter e,
                                 EVector opands,
                                 String op,
                                 int numOpands)
      throws CGException
  {
    Emittable stdin = unpackOperand(opands, numOpands);
    e.emit(stdin).put(";\n");
    e.put("synchronized␣(␣ioa.runtime.io.Stdin.class␣)\n");
    e.put("{\n");
    e.emit(stdin);
    e.put("␣=␣ioa.runtime.Automaton.getStdin();\n");
    return e;
  }
}
```

Figure 7-3: Registering the Locking Operators

`unpackOperand()` is simply a utility function of `LockOp` that checks the size of the `opands EVector` and returns its first element, cast to an `Emittable`. Note that the custom operators do not emit casts for their return values because there are none.

7.4 The Invocation Generator

One piece of the puzzle remains: where do the input files (containing lists of invocations) come from? Obviously, the user could enter them by hand; however, this would be cumbersome (especially entering the shorthand sorts), albeit simple.

Another option is to design a graphical user interface for constructing invocations. Though this is a worthwhile goal, it would be a large project and it would be challenging to find an efficient way of entering actual parameters[4].

A third option is to design an abbreviation language that may be automatically desugared into the IOA_Invocation s-expressions. Invocations might look something like:

```
(fire (actionName caseName) ((Int 1) (Bool true)))
```

7.4.1 Reusing the Simulator

I ended up choosing a variant of the last option: Ramírez-Robredo's NDR language extensions. NDR **schedule** blocks already provide a means of specifying an action and its actual parameters: **fire** statements. They already have a notation for specifying arbitrary data type instances: the IOA data type operators. We can let the user specify the list of invocations as **fire** statements in an automaton **schedule** and use the machinery of the simulator to evaluate the actual parameters.

Some benefits to this approach soon become evident:

- The user already knows IOA syntax and so should find the notation familiar.

[4]For instance, we would want a type inference mechanism so that if the user enters "1" she does not need to specify whether it is an Int, Nat, or Real.

- The IOA front-end can verify the input file, checking the spelling of action and case names, and the number and type of the actual parameters.
- The notation for specifying data types is concise and can make use of the front-end's type inference abilities. For instance, writing $\{\} \vdash 1$ is all that is needed to specify a singleton sequence, and the front-end will infer the sequence's subsort based on the action's signature.
- Temporary variables are available. It is common for invocations to use identical or similar parameters, so this is a great notational convenience.
- The IOA and NDR control flow constructs may be used; rather than writing out a straight list of **fire** statements, the user can write a *program* using loops and conditionals to generate all the **fire**s.
- Most of the needed functionality is already implemented in the front-end and the simulator. We just need to hook everything together in the right way.
- The **fire** statements for generating input may potentially be reused when developing with the simulator.

Note that the automaton passed to the invocation generator should be the algorithm automaton before it has been composed with the interface automaton. The reason is that after composition the input actions (for which the invocation generator is creating invocations) will have become internal.

7.4.2 Implementation

The invocation generator is implemented by the `ioa.codegen.ig.InvocationGenerator` class. It is a shell tool that takes as input an IL file and the number of invocations to generate. As with the simulator, if the file contains more than one automaton an automaton name may be specified on the command line to disambiguate.

Internally, the invocation generator runs a copy of the IOA simulator. It adds a listener to the simulator to receive notifications when transitions are fired. When a

transition fires, the invocation generator creates an IOA_Invocation tuple that combines the name of the action with its actual parameters. It then converts the IOA_Invocation to an s-expression and prints it to the console.

7.4.3 Example

Figure 7-4 shows an example input to the invocation generator. For simplicity, the automaton contains only two (input) transitions, and all the parameters are of type Nat.

Figure 7-5 shows the invocations outputted by the invocation generator when the schedule in Figure 7-4 is executed for eight steps. The output is a list of s-expressions that represent IOA_Invocations. I have added linebreaks and indenting to make the output easier to read and italicized the important parts. Within each tuple, the first italicized number tells the name of the action to be invoked. 0.0 denotes send and 1.0 denotes send2. The second and subsequent italicized numbers are the transition parameters.

7.4.4 Future Work

Due to lack of time, the invocation generator is more cumbersome to use than it needs to be. It requires that the input file contain an automaton and a **schedule** block that will be used to generate the invocations. In normal use, I expect that the user will have one input file for each node. At present, this requires repeating the same automaton definition at the beginning of each input file.

Ideally, the user would create a single file containing the automaton definition. It might also contain a **schedule** block to control the execution at the node (in the manner of Chapter 5). The user would then create a collection of files containing *only* **schedule** blocks. These would be used to generate the invocations for the input at each of the nodes. The invocation generator would be able to take as input the automaton file and one of the input files. To generate invocations, it would simulate the automaton using the schedule in the input file, not the one in the automaton file.

```
axioms NonDet

automaton Invocation01
  signature
    input send(n:Nat),
          send2(n1:Nat, n2:Nat)
  states
    queue: Seq[Nat] := {}
  transitions
    input send(n:Nat)
      eff queue := n ⊣ queue
    input send2(n1:Nat, n2:Nat)
      eff queue := n2 ⊣ (n1 ⊣ queue)
  schedule
    states
      a: Nat := 6
    do while true do
      fire input send(a);
      a := a + 1;
      if a > 8 then
        fire input send2(randomNat(100,200),
                         randomNat(200,300))
      else
        fire input send(1)
      fi
    od
  od
```

Figure 7-4: Input to the Invocation Generator

```
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 6)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 1)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 7)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 1)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 8)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (1.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 148)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat))))
                                     (ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 269)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (0.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 9)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
(ioa.runtime.adt.TupleSort
 ((action (ioa.runtime.adt.EnumSort (1.0 send send2)))
  (params (ioa.runtime.adt.LSeqSort ((ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 117)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat))))
                                     (ioa.runtime.adt.UnionSort
                                      ((ioa.runtime.adt.NatSort 228)
                                       (ioa.runtime.adt.EnumSort (0.0 Nat)))))))))
```

Figure 7-5: Output of the Invocation Generator

Chapter 8

Miscellany

Some changes and additions that I have made to the IOA toolkit did not belong in any of the previous chapters but are nonetheless relevant to this thesis. In this chapter I briefly discuss some of them.

8.1 Preconditions and **where** Clauses

The code emitter now verifies transition parameters and **where** clauses. Each transition translates into a Java method, and at the beginning of the method the emitter evaluates the precondition and **where** clause terms. It passes the values to methods in the base class that check whether they are true, signalling an error if they are not.

8.2 API Documentation

Most of the code-level documentation for the IOA toolkit is written in Javadoc comments. I have added a doclet (heavily based on work by the 6.170 course staff) that extends Javadoc to support additional documentation tags, such as `@requires` and `@fix`, that are used throughout the IOA codebase. A Perl script called `makedocs` automates the process of generating Javadocs for the entire IOA codebase. One challenge is that portions of the IOA front end are written in the PolyJ language, which Javadoc has difficulty parsing. `makedocs` contains some preliminary work towards

massaging PolyJ files into a format that Javadoc can read.

8.3 Regression Tests

In Section 2.5 I described unit tests for the ADT implementations and registration classes. However, it is also desirable to have end-to-end regression tests for the different components of the IOA toolkit.

I have developed a regression testing framework that is easy to use and easy to extend to support additional tests and tools as they become available. The *tools* currently supported by the framework are the static semantic checker (and IL generator), the code generator, the simulator, the simulator's Daikon connection, the paired simulator, the code generator, the IOA prettyprinter, the IOA-to-LaTeX converter, the IL unparser, and the IL-to-IOA converter.

The framework includes a library of *tests* contributed by tool developers. A test consists of an input file for a particular tool and one or more *goal* files that specify the expected output when the tool is run on the input file. If the input file is written in IOA but the tool requires input in a different format (such as IL), the framework will perform the necessary conversions.

For example, when testing the code generator, the user supplies an IOA input file and goal files for the output of the code emitter and the Java compiler. The framework uses `ioaCheck -il` to convert the IOA file to IL, which it feeds to the code generator. The code generator outputs a Java file, which the framework compares to the goal that the user has provided. In addition, the framework tries to compile the generated Java code and compares the output of the Java compiler against the list of expected compiler warnings (usually none).

The framework provides commands for controlling which tools are run with which test cases and for displaying and summarizing the results. There are also commands for adding new tests, updating the goal files with the current output, and committing updated goal files into CVS.

In addition to end-to-end tests, the regression testing framework can run the

unit tests from Section 2.5 and other unit tests that are part of the simulator. The framework may be found in the toolkit's `Test` directory in the `Makefile` and `Makefile.common` files. Running `make help` in that directory prints out the documentation for running the existing tests and adding new ones.

8.4 Graphical User Interface

Our approach of compiling I/O automata using a series of source-to-source transformers (Figure 1-1) has many advantages from a modelling standpoint. It is also convenient for tool development because the individual tools may be implemented and verified separately. One downside to this approach is that the end user of the toolkit must contend with all of the tools. Each tool requires a particular kind of input (IOA or IL) and has various command-line switches, and the tools must be used in a particular order. As a result, our system is difficult to use.

In Section 8.3 I describe one way to improve usability for toolkit developers. However, it does not go far enough for end users. I think what is needed is an easy-to-use graphical user interface (GUI) for the IOA tookit.

I have created a prototype of such a tool using Java's Swing libraries. The prototype is centered around the idea of a *project*, a collection of related files such as auxliary automata and inputs to the different nodes. The Project window shows a list of the files in the project for easy access, and projects may be saved to disk to remember the files between sessions.

The GUI lets the user edit IOA files with a rudimentary editor, and it can show the corresponding IL file, which is useful for tool debugging. The user can press a button to check the syntax of an IOA file, and the GUI's line number display helps the user locate and fix any syntax errors that are found. The user can edit either the IOA or the IL and press a button to cause the changes to be reflected in the other. Since the GUI's text editor is primitive compared to other editors—such as GNU Emacs, which can be used with an IOA mode written by Laura Dean—each editor window contains a button for opening the current file in the user's preferred external

editor.

Finally, the GUI has menu commands for running the interface generator (Chapter 3) and next action determinator (Chapter 4) tools. After one of these tools transforms the frontmost IOA window, the GUI converts the output of the tool from IL into IOA and displays it in a new window.

Of course, this is only the beginning of what the GUI could do to help the user. In [26] I suggest some ways the GUI could be extended, and I think that users of modern integrated development environments will have many ideas of their own. Although the current GUI is presently inadequate for real use, I hope that it provides a good starting point for future toolkit developers.

Chapter 9

Discussion and Future Work

This thesis has described several tools that are used in the process of generating Java code for IOA programs. Our code generation system is not yet complete, mainly because the composer tool has not yet been completed. This means that we cannot combine the algorithm automaton with the auxiliary automata necessary to connect it to the console and the network.

The tools do, however, work in isolation. The ADT library and the registry have been successfully used by researchers using the IOA simulator and its Daikon connection. This document has demonstrated the source-to-source transformers and the invocation generator. In the absence of the composer, I cannot demonstrate all the tools in this thesis working in an integrated fashion. However, it is possible to show the interaction between the ADTs and the NDR compiler, and I will do so now.

For an example IOA program I have chosen one of the banking automata that Toh Ne Win and Gustavo Santos used to test the IOA-Daikon connection [32]. Appendix C.1 shows the IOA program, complete with a schedule written using the NDR language extensions. Appendix C.2 shows the Java that the code generator created, using the registry to look up ADT implementations and employing the NDR compiler to resolve nondeterminism. Finally, Appendix C.3 shows the trace produced by running the Java code. For the reasons discussed in Section 5.7, the output is not exactly the same as that produced by the simulator. However, it does demonstrate some of the key tools developed in this thesis.

The main work that remains is the implementation of the composer tool. Once we are using the composer, the input to the code emitter will be of a slightly different form. State variables in the output of the composer are stored in tuples based on the automaton that they originated from. This is important because it will affect the way the runtime libraries access state variables. The SEND and RECEIVE actions read and write to sequence state variables that are queues of messages. In order to access the sequences, the runtime library will need to locate the `TupleSort`s (among the automaton's instance variables) that correspond to the states of the network automata.

The same challenge applies to the the interface automaton, which needs to access the stdin and stdout sequences. As discussed in Chapter 7, the translated state variables are only accessed from the main thread. The input and output threads access the sequences through methods on the base `Automaton` class (which itself does not access the state variables, either). The only references to the state variables, then, are in the code emitted by the locking operators (Section 7.3.3). The locking operators receive a reference to the target tree nodes that represent stdin and stdout. They will need to examine the structure of this operator application to find the tuple that holds the sequence state variables. Then they will be able to emit code that assigns to the state variables by setting values of the tuple's fields.

Once the code generator is fully working, we will likely want to examine ways to improve performance. In the initial implementation, all of the abstract data types are implemented using immutable Java objects. It is very inefficient to copy an entire collection each time it is modified. After composition, each automaton's state variables are collected in a single tuple. This means that changing one variable will cause a copy of the entire state tuple for that automaton. Future work should investigate how to use mutability behind the scenes to reduce unneeded copying.

Finally, users would greatly appreciate an improved graphical user interface that can guide them through the code generation process, control which machines are used to run the computation, and display the execution at each node.

Appendix A

ADT Implementation Examples

These sections show complete examples of the files one needs to create to add support for a new sort or sort constructor.

A.1 String: A Simple Sort

A.1.1 LSL Trait

This trait simply lists the operators on String; the real LSL trait also states properties of these operators.

```
String: trait
  includes
    Sequence(Char for E, String for Seq[E])
  introduces
    __<__, __≤__, __>__, __≥__: String, String → Bool

Sequence(E): trait
  includes
    Integer
  introduces
    {}: → Seq[E]
    __⊢__: Seq[E], E → Seq[E]
    __⊣__: E, Seq[E] → Seq[E]
    __‖__: Seq[E], Seq[E] → Seq[E]
    __∈__: E, Seq[E] → Bool
    head, last: Seq[E] → E
    tail, init: Seq[E] → Seq[E]
    len: Seq[E] → Int
    __[__]: Seq[E], Int → E
```

A.1.2 Implementation Class: ioa.runtime.adt.StringSort

```
/*
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.runtime.adt;

import ioa.util.sexp.*;

/**
 * <tt>StringSort</tt> implements Strings using <tt>java.lang.String</tt>.
 *
 * @author Michael Tsai (00/05/??) -- Wrapper for SeqSort[CharSort]
 * @author Michael Tsai (00/06/27) -- Rewrote to use String
 * @see ioa.registry.java.StringSort
 * @see ioa.runtime.adt.BoolSort
 * @see ioa.runtime.adt.CharSort
 * @see ioa.runtime.adt.IntSort
 */

public class StringSort extends ComparableADT implements java.io.Serializable
{
    // Code Generation Methods
    // -----------------------

    /** {}: -> String */
    public static StringSort empty() {
        return new StringSort();
    }

    /** __|-__: String, Char -> String */
    public static StringSort append(StringSort s, CharSort c) {
        return s.append(c);
    }

    /** __-|__: Char, String -> String */
    public static StringSort prepend(CharSort c, StringSort s) {
        return s.prepend(c);
    }

    /** __||__: String, String -> String */
    public static StringSort catenate(StringSort s1, StringSort s2) {
        return s1.catenate(s2);
    }

    /** __\\in__: Char, String -> Bool */
    public static BoolSort in(CharSort c, StringSort s) {
        return s.in(c);
    }

    /** head: String -> Char */
    public static CharSort head(StringSort s) {
        return s.head();
    }

    /** last: String -> Char */
    public static CharSort last(StringSort s) {
```

```
    return s.last();
}

/** tail: String -> String */
public static StringSort tail(StringSort s) {
    return s.tail();
}

/** init: String -> String */
public static StringSort init(StringSort s) {
    return s.init();
}

/** len: String -> Int */
public static IntSort len(StringSort s) {
    return s.len();
}

/** __[__]: String, Int -> Char */
public static CharSort index(StringSort s, IntSort i) {
    return s.index(i);
}

/** __<__: String, String -> Bool */
public static BoolSort lt(StringSort s1, StringSort s2) {
    return s1.lt(s2);
}

/** __<=__: String, String -> Bool */
public static BoolSort lte(StringSort s1, StringSort s2) {
    return s1.lte(s2);
}

/** __>__: String, String -> Bool */
public static BoolSort gt(StringSort s1, StringSort s2) {
    return s1.gt(s2);
}

/** __>=__: String, String -> Bool */
public static BoolSort gte(StringSort s1, StringSort s2) {
    return s1.gte(s2);
}

// Member Variables
// ----------------
protected String string;

// Creators
// --------
public StringSort() {
    this.string = "";
}

public StringSort(String string) {
    this.string = string;
}

public static ADT construct(Parameterization p)
{
    return empty();
}

// Observers
// ---------
public String toString() {
    return this.string;
}

public BoolSort in(CharSort c) {
    return BoolSort.lit(this.string.indexOf(c.toString()) != -1);
```

```
}

public IntSort len() {
    return IntSort.lit(this.string.length());
}

public CharSort index(IntSort i) {
    if ( i.value() < 0 )
        throw new RepException("Index␣given␣to␣StringSort␣was␣less␣than␣0");
    else if ( i.value() <= this.string.length() - 1)
        return new CharSort(new Character(this.string.charAt(i.value())));
    else
        throw new RepException("Can't␣take␣index␣"+i+
                               "␣because␣the␣String␣isn't␣that␣long.");
}

public int compareTo(Object o)
{
    StringSort s = (StringSort)o;
    return this.string.compareTo(s.string);
}

public BoolSort lt(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) < 0);
}

public BoolSort lte(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) <= 0);
}

public BoolSort gt(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) > 0);
}

public BoolSort gte(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) >= 0);
}

public boolean equals(Object o) {
    if ( ! (o instanceof StringSort) )
        return false;
    StringSort s = (StringSort)o;
    return this.string.equals(s.string);
}

public int hashCode() {
    return this.string.hashCode();
}

public boolean isEmpty() {
    return this.string.length() == 0;
}

// Producers
// ---------
public StringSort append(CharSort c) {
    return new StringSort(this.string + c.toString());
}

public StringSort prepend(CharSort c) {
    return new StringSort(c.toString() + this.string);
}

public StringSort catenate(StringSort s) {
    return new StringSort(this.string + s.string);
}
```

```
    public CharSort head() {
        if ( ! isEmpty() )
            return CharSort.lit(new Character(this.string.charAt(0)));
        else
            throw new
                RepException("Attempt to take head() of empty StringSort");
    }

    public CharSort last() {
        if ( ! isEmpty() )
            return CharSort.lit(
                new Character(this.string.charAt(this.string.length() - 1)));
        else
            throw new
                RepException("Attempt to take last() of empty StringSort");
    }

    public StringSort tail() {
        if ( ! isEmpty() )
            return new StringSort(
              this.string.substring(1, this.string.length()));
        else
            throw new
                RepException("Attempt to take tail() of empty StringSort");
    }

    public StringSort init() {
        if ( ! isEmpty() )
            return new StringSort(
               this.string.substring(0, this.string.length() - 1));
        else
            throw new
                RepException("Attempt to take init() of empty StringSort");
    }

    // SValue Conversions
    // ------------------

    public SValue toSValue()
    {
        SValue rep = new SString(string);
        return toSValue(rep);
    }

    public static ADT construct(SValue svalue)
    {
        return new StringSort(svalue.toString());
    }
}
```

A.1.3 Registration Class: `ioa.registry.java.StringSort`

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */
```

```
package ioa.registry.java;

import ioa.registry.ConstrImplRegistry;
import ioa.registry.ImplFactory;
import ioa.registry.Installer;
import ioa.registry.Registrable;
import ioa.registry.RegistryException;

/**
 * Registration class for Strings.
 *
 * @author Michael Tsai
 * @see ioa.runtime.adt.BoolSort
 * @see ioa.runtime.adt.CharSort
 * @see ioa.runtime.adt.IntSort
 * @see ioa.runtime.adt.StringSort
 */
public class StringSort implements Registrable
{
    // Class Variables

    /**
     * Name of the Sort for which this registers implementations.
     */
    public final static String sortName = "String";

    /**
     * Name of the Class that implements the Sort
     */
    public final static String className = "StringSort";

    /**
     * Install mappings from the sort String and its operators to the
     * ioa.runtime.adt.StringSort class and its methods in the given
     * registry.
     */
    public void install(ConstrImplRegistry reg) throws RegistryException
    {
        Installer installer =
            ImplFactory.getInstance().newInstaller(className, reg);

        installer.addSort(sortName);

        installer.addOp("({__}␣()␣%me)", "empty");
        installer.addOp("(__|-__␣(%me␣Char)␣%me)", "append");
        installer.addOp("(__-|__␣(Char␣%me)␣%me)", "prepend");
        installer.addOp("(__||__␣(%me␣%me)␣%me)", "catenate");
        installer.addOp("(__\\in__␣(Char␣%me)␣Bool)", "in");
        installer.addOp("(head␣(%me)␣Char)", "head");
        installer.addOp("(last␣(%me)␣Char)", "last");
        installer.addOp("(tail␣(%me)␣%me)", "tail");
        installer.addOp("(init␣(%me)␣%me)", "init");
        installer.addOp("(len␣(%me)␣Int)", "len");
        installer.addOp("(__[__]␣(%me␣Int)␣Char)", "index");
        installer.addOp("(__<__␣(%me␣%me)␣Bool)", "lt");
        installer.addOp("(__<=__␣(%me␣%me)␣Bool)", "lte");
        installer.addOp("(__>__␣(%me␣%me)␣Bool)", "gt");
        installer.addOp("(__>=__␣(%me␣%me)␣Bool)", "gte");
    }
}
```

A.1.4 Test Class

```
/*
 * Copyright (c) 2000 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
```

```
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.test.junit.runtime.adt;

import ioa.runtime.adt.ADT;
import ioa.runtime.adt.BoolSort;
import ioa.runtime.adt.CharSort;
import ioa.runtime.adt.IntSort;
import ioa.runtime.adt.RepException;
import ioa.runtime.adt.StringSort;
import ioa.util.sexp.*;
import junit.framework.*;

/**
 * JUnit-based black box and glass box tests for ioa.runtime.adt.StringSort.
 * @author Michael J. Tsai (00/06/27)
 */
public class StringSortTest extends TestCase
{
    // Member Variables
    // ----------------
    protected StringSort empty;
    protected StringSort aString;
    protected StringSort bString;
    protected StringSort ab;
    protected StringSort abc;
    protected StringSort abcd;
    protected StringSort bc;
    protected StringSort bcd;
    protected StringSort cd;
    protected StringSort sexp;

    protected IntSort zero;
    protected IntSort one;
    protected IntSort two;
    protected IntSort three;
    protected IntSort minusOne;

    protected CharSort a;
    protected CharSort b;
    protected CharSort c;
    protected CharSort d;

    protected BoolSort bTrue;
    protected BoolSort bFalse;

    /**
     * Runs all the tests in this class and outputs the results to stdout.
     */
    public static void main(String[] args)
    {
        junit.textui.TestRunner.run(suite());
    }

    // Framework Stuff
    // ---------------
    public StringSortTest(String name)
    {
        super(name);
    }

    /**
```

```
 * Set up the fixtures.
 */
protected void setUp()
{
    this.empty = new StringSort("");
    this.aString = new StringSort("a");
    this.bString = new StringSort("b");
    this.ab = new StringSort("ab");
    this.abc = new StringSort("abc");
    this.abcd = new StringSort("abcd");
    this.bc = new StringSort("bc");
    this.bcd = new StringSort("bcd");
    this.cd = new StringSort("cd");
    this.sexp = new StringSort("␣(foo␣2␣(3))");

    this.a = CharSort.lit(new Character('a'));
    this.b = CharSort.lit(new Character('b'));
    this.c = CharSort.lit(new Character('c'));
    this.d = CharSort.lit(new Character('d'));

    this.zero = IntSort.lit(0);
    this.one = IntSort.lit(1);
    this.two = IntSort.lit(2);
    this.three = IntSort.lit(3);
    this.minusOne = IntSort.lit(-1);

    this.bFalse = BoolSort.False();
    this.bTrue = BoolSort.True();
}

/**
 * @return a single Test that runs all the tests in this class
 */
public static Test suite()
{
    return new TestSuite(StringSortTest.class);
}

// Test Methods
// ------------
public void testEmpty()
{
    assertEquals("", empty.toString());
}

public void testAppend()
{
    assertEquals(abc, ab.append(c));
    assertEquals(abcd, abc.append(d));
    assertEquals("a", empty.append(a).toString());
}

public void testPrepend()
{
    assertEquals(abc, bc.prepend(a));
    assertEquals(abcd, bcd.prepend(a));
    assertEquals("a", empty.prepend(a).toString());
}

public void testCatenate()
{
    assertEquals(empty, empty.catenate(empty));
    assertEquals(ab, empty.catenate(ab));
    assertEquals(ab, ab.catenate(empty));
    assertEquals(abcd, ab.catenate(cd));
}

public void testIn()
{
    assertEquals(bTrue, ab.in(a));
```

```
    assertEquals(bTrue, abc.in(a));
    assertEquals(bTrue, ab.in(b));
    assertEquals(bTrue, abc.in(b));
    assertEquals(bFalse, ab.in(c));
    assertEquals(bTrue, abc.in(c));
    assertEquals(bFalse, ab.in(c));
    assertEquals(bFalse, empty.in(a));
}

public void testHead()
{
    assertEquals(a, ab.head());
    assertEquals(a, abc.head());
    assertEquals(b, bc.head());
    try { empty.head(); fail(); } catch ( RepException e ) {}
}

public void testLast()
{
    assertEquals(b, ab.last());
    assertEquals(c, abc.last());
    assertEquals(c, bc.last());
    try { empty.last(); fail(); } catch ( RepException e ) {}
}

public void testTail()
{
    try { empty.tail(); fail(); } catch ( RepException e ) {}
    assertEquals(empty, aString.tail());
    assertEquals(bString, ab.tail());
    assertEquals(bc, abc.tail());
    assertEquals(bcd, abcd.tail());
}

public void testInit()
{
    try { empty.init(); fail(); } catch ( RepException e ) {}
    assertEquals(empty, aString.init());
    assertEquals(aString, ab.init());
    assertEquals(ab, abc.init());
    assertEquals(abc, abcd.init());
}

public void testLen()
{
    assertEquals(zero, empty.len());
    assertEquals(one, aString.len());
    assertEquals(two, ab.len());
}

public void testIndex()
{
    try { empty.index(zero); fail(); } catch ( RepException e ) {}
    try { aString.index(one); fail(); } catch ( RepException e ) {}
    try { aString.index(minusOne); fail(); } catch ( RepException e ) {}
    assertEquals(a, aString.index(zero));
    assertEquals(b, abc.index(one));
}

public void testLt()
{
    assertEquals(bTrue, aString.lt(ab));
    assertEquals(bFalse, ab.lt(aString));
    assertEquals(bTrue, ab.lt(bc));
    assertEquals(bFalse, bc.lt(ab));
    assertEquals(bFalse, ab.lt(ab));
    assertEquals(bTrue, empty.lt(ab));
}

public void testLte()
```

```
    {
        assertEquals(bTrue, aString.lte(ab));
        assertEquals(bFalse, ab.lte(aString));
        assertEquals(bTrue, ab.lte(bc));
        assertEquals(bFalse, bc.lte(ab));
        assertEquals(bTrue, ab.lte(ab));
        assertEquals(bTrue, empty.lte(ab));
    }

    public void testGt()
    {
        assertEquals(bFalse, aString.gt(ab));
        assertEquals(bTrue, ab.gt(aString));
        assertEquals(bFalse, ab.gt(bc));
        assertEquals(bTrue, bc.gt(ab));
        assertEquals(bFalse, ab.gt(ab));
        assertEquals(bFalse, empty.gt(ab));
    }

    public void testGte()
    {
        assertEquals(bFalse, aString.gte(ab));
        assertEquals(bTrue, ab.gte(aString));
        assertEquals(bFalse, ab.gte(bc));
        assertEquals(bTrue, bc.gte(ab));
        assertEquals(bTrue, ab.gte(ab));
    }

    public void testEquals()
    {
        assertEquals(bTrue, StringSort.equals(abc, new StringSort("abc")));
        assertEquals(bFalse, StringSort.equals(abc, bc));
    }

    public void testNotEquals()
    {
        assertEquals(bFalse, StringSort.notEquals(abc, new StringSort("abc")));
        assertEquals(bTrue, StringSort.notEquals(abc, bc));
    }

    public void testIfThenElse()
    {
        assertEquals(abc, StringSort.ifThenElse(bTrue, abc, bc));
        assertEquals(bc, StringSort.ifThenElse(bFalse, abc, bc));
    }

    public void testHashCode()
    {
         assertEquals(ab.hashCode(), aString.catenate(bString).hashCode());
         assertTrue(ab.hashCode() != aString.hashCode());
    }

    public void testCompareTo()
    {
        assertTrue( ab.compareTo(bc) < 0 );
        assertTrue( abc.compareTo(aString) > 0);
        assertTrue( abc.compareTo(abc) == 0);
    }

    public void testToSValue()
    {
        String expected = "(ioa.runtime.adt.StringSort␣\"␣(foo␣2␣(3))\")";
        assertEquals(expected, sexp.toSValue().toString());
    }

    public void testConstructSValue()
    {
        assertEquals(sexp, ADT.construct(sexp.toSValue()));
    }
}
```

A.1.5 IOA File

```
automaton String01
  signature
    internal a1
    states
      s: String := {},
      i: Int,
      c: Char := 'A',
      b: Bool
    transitions
      internal a1
        pre
          s = {}
        eff
          s := s ⊢ c;
          s := c ⊣ s;
          s := s ‖ s;
          b := c ∈ s;
          c := head(s);
          c := last(s);
          s := tail(s);
          s := init(s);
          i := len(s);
          c := s[2];
          b := s = s;
          b := s[1] = s[2];
          b := s < s;
          b := s ≤ s;
          b := s > s;
          b := s ≥ s;
          b := s = s;
          b := s ≠ s;
          s := if b then s else s
```

A.1.6 Generated Java Code

```
import java.io.*; import ioa.runtime.adt.*; import ioa.runtime.*;

public class String01 extends ioa.runtime.Automaton {
  public StringSort s_v1 = ((StringSort)StringSort.empty());
  public IntSort i_v2 = (IntSort)IntSort.construct(new Parameterization());
  public CharSort c_v3 = ((CharSort)CharSort.lit('A'));
  public BoolSort b_v4 =
    (BoolSort)BoolSort.construct(new Parameterization());
  public void action_a1()
    {
      enteredTransition("internal a1(" + ") in automaton String01");
      assertPrecondition((((BoolSort)StringSort.equals(s_v1,

((StringSort)StringSort.empty()))))).booleanValue());
      s_v1 = ((StringSort)StringSort.append(s_v1, c_v3));
      s_v1 = ((StringSort)StringSort.prepend(c_v3, s_v1));
      s_v1 = ((StringSort)StringSort.catenate(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.in(c_v3, s_v1));
      c_v3 = ((CharSort)StringSort.head(s_v1));
      c_v3 = ((CharSort)StringSort.last(s_v1));
```

```
      s_v1 = ((StringSort)StringSort.tail(s_v1));
      s_v1 = ((StringSort)StringSort.init(s_v1));
      i_v2 = ((IntSort)StringSort.len(s_v1));
      c_v3 = ((CharSort)StringSort.index(s_v1, ((IntSort)IntSort.lit(2))));
      b_v4 = ((BoolSort)StringSort.equals(s_v1, s_v1));
      b_v4 =
        ((BoolSort)CharSort.equals(((CharSort)StringSort.index(s_v1,

((IntSort)IntSort.lit(1)))),
                                   ((CharSort)StringSort.index(s_v1,

((IntSort)IntSort.lit(2))))));
      b_v4 = ((BoolSort)StringSort.lt(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.lte(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.gt(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.gte(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.equals(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.notEquals(s_v1, s_v1));
      s_v1 = ((StringSort)((b_v4).booleanValue() ? s_v1 : s_v1));
      exitedTransition("internal␣a1(" + ")␣in␣automaton␣String01");

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"String01"});
}

}
```

A.2 Set: A Compound Sort

A.2.1 LSL Trait

This trait simply lists the operators on Set; the real LSL trait also states properties of these operators. It is not shown here because it assumes other traits, which makes it harder to see what all the operators are.

```
Set(E): trait
  includes
    Integer
  introduces
    {}: → Set[E]
    {__}: E → Set[E]
    insert, delete: E, Set[E] → Set[E]
    __∈__: E, Set[E] → Bool
    __∪__, __∩__, __−__: Set[E], Set[E] → Set[E]
    __⊂__, __⊃__, __⊆__, __⊇__: Set[E], Set[E] → Bool
    size: Set[E] → Int
```

A.2.2 Implementation Class: `ioa.runtime.adt.SetSort`

```
/*
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
```

```
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.runtime.adt;

import ioa.util.sexp.*;
import ioa.util.logger.IOACategory;
import ioa.util.ToStringComparator;

import java.util.Collections;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

/**
 * <p> <tt>SetSort</tt> implements IOA sets using a <tt>HashSet</tt> for
 * speed. Because it was written before <tt>MsetSort</tt>, it is not
 * based on <tt>MsetSort</tt>. Also, it is probably faster this way.</p>
 *
 * @author Michael Tsai (00/04/19)
 * @see ioa.registry.java.SetSort
 * @see ioa.runtime.adt.IntSort
 * @see ioa.runtime.adt.BoolSort
 */
public class SetSort extends ADT implements Cloneable,
                                             Enumerable,
                                             java.io.Serializable
{
    // Class Variables
    private static IOACategory
    cat = IOACategory.getInstance(SetSort.class.getName());

    // Member Variables
    // ----------------

    protected Set set = new HashSet();

    // Creators
    // --------

    /** Construct a new, empty set. */
    public SetSort() {}

    // Code Generation Methods
    // -----------------------

    /** {}:  -> Set[E] */
    public static SetSort empty() {
        return new SetSort();
    }

    /** {__}: E -> Set[E] */
    public static SetSort singleton(Object o) {
        return new SetSort().insert(o);
    }

    /** insert: E, Set[E] -> Set[E] */
    public static SetSort insert(Object o, SetSort s) {
        return s.insert(o);
    }
```

```
/** delete; E, Set[E] -> Set[E] */
public static SetSort delete(Object o, SetSort s) {
    return s.delete(o);
}

/** __\in__: E, Set[E] -> Bool */
public static BoolSort isIn(Object o, SetSort s) {
    return BoolSort.lit(s.contains(o));
}

/** __\U__: Set[E], Set[E] -> Set[E] */
public static SetSort union(SetSort s1, SetSort s2) {
    return s1.union(s2);
}

/** __\I__: Set[E], Set[E] -> Set[E] */
public static SetSort intersection(SetSort s1, SetSort s2) {
    return s1.intersection(s2);
}

/** __-__: Set[E], Set[E] -> Set[E] */
public static SetSort difference(SetSort s1, SetSort s2) {
    return s1.difference(s2);
}

/** __\supset__: Set[E], Set[E] -> Bool */
public static BoolSort isSupset(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSupset(s2));
}

/** __\subset__: Set[E], Set[E] -> Bool */
public static BoolSort isSubset(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSubset(s2));
}

/** __\subseteq__: Set[E], Set[E] -> Bool */
public static BoolSort isSubsetEq(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSubsetEq(s2));
}

/** __\supseteq__: Set[E], Set[E] -> Bool */
public static BoolSort isSupsetEq(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSupsetEq(s2));
}

/** size: Set[E] -> Int */
public static IntSort size(SetSort s) {
    return IntSort.lit(s.size());
}

// Observers
// ---------

/** @return true if this contains o and false otherwise */
public boolean contains(Object o) {
    return set.contains(o);
}

/** @return true if this \subset s and false otherwise */
public boolean isSubset(SetSort s) {
    return s.set.containsAll(this.set) && ! this.set.equals(s.set);
}

/** @return true if this \supset s and false otherwise */
public boolean isSupset(SetSort s) {
    return this.set.containsAll(s.set) && ! this.set.equals(s.set);
}

/** @return true if this \subseteq s and false otherwise */
public boolean isSubsetEq(SetSort s) {
```

```
    return s.set.containsAll(this.set);
}

/** @return true if this \supseteq s and false otherwise */
public boolean isSupsetEq(SetSort s) {
    return this.set.containsAll(s.set);
}

/** @return |this| */
public int size() {
    return set.size();
}

/**
 * Returns the underlying java.util.Set, but
 * the elements are unmodifiable.  The elements
 * are also ordered alphabetically as they'd print out.
 **/

public Set getSet() {
  return Collections.unmodifiableSet(set);
}

/**
 * Returns the underlying java.util.Set, but the elements are
 * unmodifiable.  The elements are also ordered alphabetically as
 * they'd print out - this should only be used for diff testing to
 * remove nondeterminism in printing.
 *
 **/

public Set getSetOrdered() {
  TreeSet ts = new TreeSet (new ToStringComparator());
  ts.addAll (set);
  return Collections.unmodifiableSet(ts);
}

public boolean equals(Object o) {
    if ( o instanceof SetSort )
        return this.set.equals(((SetSort) o).set);
    else
        return false;
}

public int hashCode() {
    int result = 0;

    Iterator iter = this.set.iterator();
    while ( iter.hasNext() )
    {
        result += iter.next().hashCode();
    }

    return result;
}

// Producers
// --------

/** @return shallow copy of this */
public Object clone() {
    SetSort result = new SetSort();
    result.set.addAll(this.set);
    return result;
}

/** @return this \U {o} */
public SetSort insert(Object o) {
    SetSort result = (SetSort)this.clone();
    result.set.add(o);
```

```
    return result;
}

/** @return this - {o} */
public SetSort delete(Object o) {
    SetSort result = (SetSort)this.clone();
    result.set.remove(o);
    return result;
}

/** @return a new SetSort whose elements are this \U s */
public SetSort union(SetSort s) {
    SetSort result = (SetSort)this.clone();
    result.set.addAll(s.set);
    return result;
}

/** @return a new SetSort whose elements are this \I s */
public SetSort intersection(SetSort s) {
    SetSort result = (SetSort)this.clone();
    result.set.retainAll(s.set);
    return result;
}

/** @return a new SetSort whose elements are this - s */
public SetSort difference(SetSort s) {
    SetSort result = (SetSort)this.clone();
    result.set.removeAll(s.set);
    return result;
}

/**
 * Select a random element from a set.
 * @return a random element of this set.
 * @exception RepException if the set is empty
 **/
public static Object chooseRandom (SetSort set) {
    int i = set.size();
    if (i == 0)
        throw new
            RepException("Cannot␣take␣an␣element␣out␣of␣an␣empty␣set");
    return set.getSet().toArray()[NonDet.rnd.nextInt (i)];
}

/**
 * Exclude a random element from a set and return the rest.
 * @return a subset of the set such that there's one element removed.
 * @exception RepException if the set is empty
 **/
public static SetSort rest (SetSort set) {
    int i = set.size();
    if (i == 0)
        throw new
            RepException("Cannot␣take␣an␣element␣out␣of␣an␣empty␣set");
    return SetSort.delete(set.getSet().toArray()[NonDet.rnd.nextInt (i)],
                          set);
}

/**
 * Predicate on whether a set is empty
 * @return true of the set is empty.
 **/
public static BoolSort isEmpty (SetSort set) {
    int i = set.size();
    return BoolSort.lit (i == 0);
}

/** @return an instance of SetSort */
public static ADT construct(Parameterization p)
{
```

```
        return empty();
    }

    /**
     * Returns a String representation of this, in alphabetical order
     * @return a String representation of this.
     **/
    public String toString() {
        Iterator iter = getSetOrdered().iterator();
        String result = "(";

        while ( iter.hasNext() ) {
            result += iter.next().toString() + "␣";
        }
        return result.trim() + ")";
    }

    // Enumerable
    // ----------
    public Iterator iterator()
    {
        return set.iterator();
    }


    // SValue Conversions
    // ------------------

    public SValue toSValue()
    {
        SList rep = new SList();
        // don't use getSetOrdered() if we care about performance
        Iterator i = getSetOrdered().iterator();
        while ( i.hasNext() )
        {
            ADT adt = (ADT)i.next();
            rep.add(adt.toSValue());
        }
        return toSValue(rep);
    }

    public static ADT construct(SValue svalue)
    {
        SetSort set = empty();
        SList rep = (SList)svalue;
        Iterator i = rep.iterator();
        while ( i.hasNext() )
        {
            SValue s = (SValue)i.next();
            set.set.add(ADT.construct(s));
        }
        return set;
    }
}
```

A.2.3 Registration Class: ioa.registry.java.SetSort

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
```

```
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.registry.java;

import ioa.registry.ConstrImplRegistry;
import ioa.registry.ImplFactory;
import ioa.registry.Installer;
import ioa.registry.Registrable;
import ioa.registry.RegistryException;

/**
 * Registration class for Sets. The implementations of
 * ioa.registry.java.SetSort and ioa.runtime.adt.SetSort are
 * intertwined. Changes in one should be reflected in the other.
 *
 * @author Michael Tsai (00/04/20)
 * @see ioa.runtime.adt.SetSort
 */
public class SetSort implements Registrable
{
    // Class Variables

    /** Name of the Sort for which this registers implementations. */
    public final static String sortName = "Set";

    /** Name of the Class that implements the Sort */
    public final static String className = "SetSort";


    /**
     * Install mappings from the Sort constructor Set and its operators
     * to the ioa.runtime.adt.SetSort class and its methods in the given
     * registry.
     */
    public void install(ConstrImplRegistry reg) throws RegistryException
    {
        Installer installer =
            ImplFactory.getInstance().newInstaller(className, reg);

        installer.addSort("(Set␣0)");

        installer.addOp("({__}␣()␣%me)", "empty");
        installer.addOp("({__}␣(0)␣%me)", "singleton");
        installer.addOp("(insert␣(0␣%me)␣%me)", "insert");
        installer.addOp("(delete␣(0␣%me)␣%me)", "delete");
        installer.addOp("(__\\in__␣(0␣%me)␣Bool)", "isIn");
        installer.addOp("(__\\U__␣(%me␣%me)␣%me)", "union");
        installer.addOp("(__\\I__␣(%me␣%me)␣%me)", "intersection");
        installer.addOp("(__-__␣(%me␣%me)␣%me)", "difference");
        installer.addOp("(__\\subset__␣(%me␣%me)␣Bool)", "isSubset");
        installer.addOp("(__\\supset__␣(%me␣%me)␣Bool)", "isSupset");
        installer.addOp("(__\\subseteq__␣(%me␣%me)␣Bool)", "isSubsetEq");
        installer.addOp("(__\\supseteq__␣(%me␣%me)␣Bool)", "isSupsetEq");
        installer.addOp("(size␣(%me)␣Int)", "size");
        installer.addOp("(chooseRandom␣(%me)␣0)", "chooseRandom");
        installer.addOp("(rest␣(%me)␣%me)", "rest");
        installer.addOp("(isEmpty␣(%me)␣Bool)", "isEmpty");
    }
}
```

A.2.4 Test Class

```
/*
 * Copyright (c) 2002 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
```

```
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.test.junit.runtime.adt;

import junit.framework.*;
import ioa.runtime.adt.SetSort;
import ioa.runtime.adt.BoolSort;
import ioa.runtime.adt.IntSort;
import ioa.runtime.adt.ADT;
import ioa.util.sexp.*;

/**
 * JUnit-based black box and glass box tests for ioa.runtime.adt.SetSort.
 * @author Michael J. Tsai (00/06/22)
 */
public class SetSortTest extends TestCase
{
    // Member Variables
    // ----------------
    protected SetSort empty;
    protected SetSort one;
    protected SetSort two;
    protected SetSort three;
    protected SetSort oneAndTwo;
    protected SetSort oneAndThree;
    protected SetSort twoAndThree;
    protected SetSort oneAndTwoAndThree;

    protected BoolSort bTrue;
    protected BoolSort bFalse;

    /**
     * Runs all the tests in this class and outputs the results to stdout.
     */
    public static void main(String[] args)
    {
        junit.textui.TestRunner.run(suite());
    }

    // Framework Stuff
    // ---------------
    public SetSortTest(String name)
    {
        super(name);
    }

    /**
     * Set up the fixtures.
     */
    protected void setUp()
    {
        empty               = new SetSort();
        one                 = empty.insert(IntSort.lit(1));
        two                 = empty.insert(IntSort.lit(2));
        three               = empty.insert(IntSort.lit(3));
        oneAndTwo           = one.insert(IntSort.lit(2));
        oneAndThree         = one.insert(IntSort.lit(3));
        twoAndThree         = two.insert(IntSort.lit(3));
        oneAndTwoAndThree   = oneAndTwo.insert(IntSort.lit(3));

        bTrue = BoolSort.True();
```

```
    bFalse = BoolSort.False();
}

/**
 * @return a single Test that runs all the tests in this class
 */
public static Test suite()
{
    return new TestSuite(SetSortTest.class);
}

// Test Methods
// ------------
public void testContains()
{
    assertTrue(oneAndThree.contains(IntSort.lit(1)));
    assertTrue(oneAndThree.contains(IntSort.lit(3)));
    assertTrue(! oneAndThree.contains(IntSort.lit(2)));
    assertTrue(! empty.contains(IntSort.lit(1)));
}

public void testIsSubset()
{
    assertTrue(empty.isSubset(one));
    assertTrue(one.isSubset(oneAndTwo));
    assertTrue(oneAndTwo.isSubset(oneAndTwoAndThree));

    assertTrue(! one.isSubset(empty));
    assertTrue(! oneAndTwo.isSubset(one));
    assertTrue(! oneAndTwoAndThree.isSubset(oneAndTwo));

    assertTrue(! oneAndTwo.isSubset(oneAndTwo));
    assertTrue(! empty.isSubset(empty));
}

public void testIsSupset()
{
    assertTrue(one.isSupset(empty));
    assertTrue(oneAndTwo.isSupset(one));
    assertTrue(oneAndTwoAndThree.isSupset(oneAndTwo));

    assertTrue(! empty.isSupset(one));
    assertTrue(! one.isSupset(oneAndTwo));
    assertTrue(! oneAndTwo.isSupset(oneAndTwoAndThree));

    assertTrue(! oneAndTwo.isSupset(oneAndTwo));
    assertTrue(! empty.isSupset(empty));
}

public void testIsSubsetEq()
{
    assertTrue(empty.isSubsetEq(one));
    assertTrue(one.isSubsetEq(oneAndTwo));
    assertTrue(oneAndTwo.isSubsetEq(oneAndTwoAndThree));

    assertTrue(! one.isSubsetEq(empty));
    assertTrue(! oneAndTwo.isSubsetEq(one));
    assertTrue(! oneAndTwoAndThree.isSubsetEq(oneAndTwo));

    assertTrue(oneAndTwo.isSubsetEq(oneAndTwo));
    assertTrue(empty.isSubsetEq(empty));
}

public void testIsSupsetEq()
{
    assertTrue(one.isSupsetEq(empty));
    assertTrue(oneAndTwo.isSupsetEq(one));
    assertTrue(oneAndTwoAndThree.isSupsetEq(oneAndTwo));

    assertTrue(! empty.isSupsetEq(one));
```

```
    assertTrue(! one.isSupsetEq(oneAndTwo));
    assertTrue(! oneAndTwo.isSupsetEq(oneAndTwoAndThree));

    assertTrue(oneAndTwo.isSupsetEq(oneAndTwo));
    assertTrue(empty.isSupsetEq(empty));
}

public void testSize()
{
    assertEquals(0, empty.size());
    assertEquals(1, one.size());
    assertEquals(2, oneAndTwo.size());
}

public void testEquals()
{
    assertTrue(oneAndTwo.equals(oneAndTwo));

    SetSort oneAndTwo2 = new SetSort();
    oneAndTwo2 = oneAndTwo2.insert(IntSort.lit(1));
    oneAndTwo2 = oneAndTwo2.insert(IntSort.lit(2));
    assertTrue(oneAndTwo.equals(oneAndTwo2));

    assertTrue(oneAndTwo.equals(oneAndTwoAndThree.delete(IntSort.lit(3))));

    assertTrue(! one.equals(two));
    assertTrue(! oneAndTwo.equals(oneAndThree));

    assertTrue(! one.equals(IntSort.lit(1)));
}

public void testNotEquals()
{
    assertEquals(bTrue, SetSort.notEquals(oneAndTwo, oneAndThree));
    assertEquals(bFalse, SetSort.notEquals(oneAndTwo, oneAndTwo));
}

public void testIfThenElse()
{
    assertEquals(oneAndTwo,
                 SetSort.ifThenElse(bTrue, oneAndTwo, oneAndThree));
    assertEquals(oneAndThree,
                 SetSort.ifThenElse(bFalse, oneAndTwo, oneAndThree));
}

public void testInsert()
{
    assertEquals(oneAndTwo, one.insert(IntSort.lit(2)));
    assertEquals("Immutability", empty.insert(IntSort.lit(1)), one);
    assertEquals(oneAndTwoAndThree, oneAndTwo.insert(IntSort.lit(3)));
    assertEquals("Immutabiliity", one.insert(IntSort.lit(2)), oneAndTwo);
}

public void testDelete()
{
    assertEquals(one, oneAndTwo.delete(IntSort.lit(2)));
    assertEquals("Immutability", one.insert(IntSort.lit(2)), oneAndTwo);
    assertEquals(oneAndTwo, oneAndTwoAndThree.delete(IntSort.lit(3)));
    assertEquals("Immutability", oneAndTwoAndThree,
                                 oneAndTwo.insert(IntSort.lit(3)));
    assertEquals(empty, empty.delete(IntSort.lit(2)));
}

public void testUnion()
{
    assertEquals(empty, empty.union(empty));
    assertEquals(one, empty.union(one));
    assertEquals(one, one.union(empty));
    assertEquals(oneAndTwo, one.union(two));
    assertEquals(oneAndTwo, two.union(one));
```

```
        assertEquals(oneAndTwoAndThree, oneAndTwo.union(three));
        assertEquals(oneAndTwoAndThree, three.union(oneAndTwo));
    }

    public void testIntersection()
    {
        assertEquals(one, oneAndTwo.intersection(oneAndThree));
        assertEquals(one, oneAndThree.intersection(oneAndTwo));
        assertEquals(two, oneAndTwo.intersection(twoAndThree));
        assertEquals(two, twoAndThree.intersection(oneAndTwo));
        assertEquals(empty, empty.intersection(one));
        assertEquals(empty, one.intersection(empty));
    }

    public void testDifference()
    {
        assertEquals(one, oneAndTwo.difference(two));
        assertEquals(one, oneAndTwoAndThree.difference(twoAndThree));
        assertEquals(one, one.difference(empty));
        assertEquals(one, one.difference(two));
        assertEquals(empty, empty.difference(empty));
    }

    public void testHashCode()
    {
        assertEquals(oneAndTwo.hashCode(), one.insert(two).hashCode());
        assertTrue(one.hashCode() != oneAndTwo.hashCode());
    }

    public void testToSValue()
    {
        String expected = "(ioa.runtime.adt.SetSort␣(" +
                          "(ioa.runtime.adt.IntSort␣1)␣" +
                          "(ioa.runtime.adt.IntSort␣2)))";
        assertEquals(expected, oneAndTwo.toSValue().toString());
    }

    public void testConstructSValue()
    {
        assertEquals(oneAndTwo, ADT.construct(oneAndTwo.toSValue()));
    }
}
```

A.2.5 IOA File

```
automaton Set01
  signature
    internal a1
  states
    s: Set[Int],
    i: Int,
    b: Bool
  transitions
    internal a1
      eff
        s := {};
        s := {3};
        s := insert(i, s);
        s := delete(i, s);
        b := i ∈ s;
        s := s ∪ s;
        s := s ∩ s;
        s := s − s;
```

```
b := s ⊂ s;
b := s ⊆ s;
b := s ⊃ s;
b := s ⊇ s;
i := size(s);
b := s = s;
b := s ≠ s;
s := if b then s else s
```

A.2.6 Generated Java Code

```
import java.io.*; import ioa.runtime.adt.*; import ioa.runtime.*;

public class Set01 extends ioa.runtime.Automaton {
  public SetSort s_v1 =
    (SetSort)SetSort.construct(new
                          Parameterization(new Class[]
                                           {ioa.runtime.adt.IntSort.class},
                                           new Parameterization[]
                                             {new
                                                Parameterization()}));
  public IntSort i_v2 = (IntSort)IntSort.construct(new Parameterization());
  public BoolSort b_v3 =
    (BoolSort)BoolSort.construct(new Parameterization());
  public void action_a1()
    {
      enteredTransition("internal␣a1(" + ")␣in␣automaton␣Set01");
      s_v1 = ((SetSort)SetSort.empty());
      s_v1 = ((SetSort)SetSort.singleton(((IntSort)IntSort.lit(3))));
      s_v1 = ((SetSort)SetSort.insert(i_v2, s_v1));
      s_v1 = ((SetSort)SetSort.delete(i_v2, s_v1));
      b_v3 = ((BoolSort)SetSort.isIn(i_v2, s_v1));
      s_v1 = ((SetSort)SetSort.union(s_v1, s_v1));
      s_v1 = ((SetSort)SetSort.intersection(s_v1, s_v1));
      s_v1 = ((SetSort)SetSort.difference(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSubset(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSubsetEq(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSupset(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSupsetEq(s_v1, s_v1));
      i_v2 = ((IntSort)SetSort.size(s_v1));
      b_v3 = ((BoolSort)SetSort.equals(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.notEquals(s_v1, s_v1));
      s_v1 = ((SetSort)((b_v3).booleanValue() ? s_v1 : s_v1));
      exitedTransition("internal␣a1(" + ")␣in␣automaton␣Set01");

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"Set01"});
}

}
```

Appendix B

NDR Compilation Example

These sections show an example illustrating the compilation process described in Section 5.

B.1 Automaton with NDR Annotations

```
automaton NDR01
  signature
    internal foo
    internal bar(i, j :Int)
  states
    n: Nat := 1,
    b: Bool := true,
    k: Int := 8
  transitions
    internal foo
      eff
        n := choose det do
                          yield 1;
                          if b then
                            yield 2;
                            yield 3
                          else
                            yield 4;
                            yield 5
                          fi;
                          while k < 10 do
                            yield 6;
                            if b then
                              yield 7;
                              yield 8
                            elseif k < 12 then
```

```
                              yield 9;
                              yield 10
                            else
                              yield 11;
                              yield 12
                            fi;
                            yield 13
                          od
                        od
    internal bar(i, i+1 :Int)
      eff
        b := choose det do
                          yield ¬b
                        od;
        k := choose det do
                          yield k + i
                        od
  schedule
    states
      baz : Bool
    do
       while true do
        fire internal foo;
        fire internal bar(1, 2);
        if baz then
          fire internal bar(k, k+1);
          baz := ¬baz
        fi
       od
    od
```

B.2 Generated Java Code

```
import java.io.*; import ioa.runtime.adt.*; import ioa.runtime.*;

public class NDR01 extends ioa.runtime.Automaton {
  public NatSort n_v4 = ((NatSort)NatSort.lit(1));
  public BoolSort b_v5 = ((BoolSort)BoolSort.True());
  public IntSort k_v6 = ((IntSort)IntSort.lit(8));
   int choosePC0 = 0;
   int choosePC1 = 17;
   int choosePC2 = 19;
  public BoolSort baz_v10 =
    (BoolSort)BoolSort.construct(new Parameterization());
   int schedulePC = 21;
  public void action_foo()
    {
      enteredTransition("internal␣foo␣in␣automaton␣NDR01");
      n_v4 = (NatSort)resolveChoose0();
      exitedTransition("internal␣foo␣in␣automaton␣NDR01");

  }

  public void action_bar( IntSort i_v1,  IntSort dummy0)
    {
      enteredTransition("internal␣bar␣in␣automaton␣NDR01");
      b_v5 = (BoolSort)resolveChoose1(i_v1, dummy0);
```

```
      k_v6 = (IntSort)resolveChoose2(i_v1, dummy0);
      exitedTransition("internal␣bar␣in␣automaton␣NDR01");

  }

  protected NatSort resolveChoose0()
    {
      while ( true )
        {
          switch ( choosePC0 ) {
              case 0: /* while loop test */
              {
                if
                (((BoolSort)BoolSort.True()).booleanValue())
                { choosePC0 = 1; break;
                }

                else
                { choosePC0 = -42; break;
                }
                ;

              }

              case 1: /* yield */
              { choosePC0 = 2; return ((NatSort)NatSort.lit(1));
              }

              case 2: /* conditional test */
              {
                if
                (b_v5.booleanValue())
                { choosePC0 = 6; break;
                }

                else
                { choosePC0 = 4; break;
                }
                ;

              }

              case 6: /* yield */
              { choosePC0 = 7; return ((NatSort)NatSort.lit(2));
              }

              case 7: /* yield */
              { choosePC0 = 3; return ((NatSort)NatSort.lit(3));
              }

              case 4: /* yield */
              { choosePC0 = 5; return ((NatSort)NatSort.lit(4));
              }

              case 5: /* yield */
              { choosePC0 = 3; return ((NatSort)NatSort.lit(5));
              }

              case 3: /* while loop test */
              {
                if
                (((BoolSort)IntSort.lt(k_v6,

((IntSort)IntSort.lit(10)))).booleanValue())
                { choosePC0 = 8; break;
                }

                else
                { choosePC0 = 0; break;
                }
```

```
                ;

              }

              case 8: /* yield */
              { choosePC0 = 9; return ((NatSort)NatSort.lit(6));
              }

              case 9: /* conditional test */
              {
                if
                (b_v5.booleanValue())
                { choosePC0 = 13; break;
                }
                else
                if
                (((BoolSort)IntSort.lt(k_v6,

((IntSort)IntSort.lit(12)))).booleanValue())
                { choosePC0 = 15; break;
                }

                else
                { choosePC0 = 11; break;
                }
                ;

              }

              case 13: /* yield */
              { choosePC0 = 14; return ((NatSort)NatSort.lit(7));
              }

              case 14: /* yield */
              { choosePC0 = 10; return ((NatSort)NatSort.lit(8));
              }

              case 15: /* yield */
              { choosePC0 = 16; return ((NatSort)NatSort.lit(9));
              }

              case 16: /* yield */
              { choosePC0 = 10; return ((NatSort)NatSort.lit(10));
              }

              case 11: /* yield */
              { choosePC0 = 12; return ((NatSort)NatSort.lit(11));
              }

              case 12: /* yield */
              { choosePC0 = 10; return ((NatSort)NatSort.lit(12));
              }

              case 10: /* yield */
              { choosePC0 = 3; return ((NatSort)NatSort.lit(13));
              }
              default:
                throw new Error("NDR program jumped to invalid PC");};

          }
          ;

  }

  protected BoolSort resolveChoose1( IntSort i_v1,  IntSort dummy0)
    {
      while ( true )
        {
          switch ( choosePC1 ) {
              case 17: /* while loop test */
```

```
            {
              if
              (((BoolSort)BoolSort.True()).booleanValue())
              { choosePC1 = 18; break;
              }

              else
              { choosePC1 = -42; break;
              }
              ;

            }

            case 18: /* yield */
            { choosePC1 = 17; return ((BoolSort)BoolSort.not(b_v5));
            }
            default:
              throw new Error("NDR program jumped to invalid PC");};

        }
        ;

}

protected IntSort resolveChoose2( IntSort i_v1,  IntSort dummy0)
  {
    while ( true )
      {
        switch ( choosePC2 ) {
            case 19: /* while loop test */
            {
              if
              (((BoolSort)BoolSort.True()).booleanValue())
              { choosePC2 = 20; break;
              }

              else
              { choosePC2 = -42; break;
              }
              ;

            }

            case 20: /* yield */
            { choosePC2 = 19; return ((IntSort)IntSort.add(k_v6, i_v1));
            }
            default:
              throw new Error("NDR program jumped to invalid PC");};

        }
        ;

}

protected void schedule()
  {
    scheduleLoop: while ( true )
      {
        switch ( schedulePC ) {
            case 21: /* while loop test */
            {
              if
              (((BoolSort)BoolSort.True()).booleanValue())
              { schedulePC = 23; break;
              }

              else
              { schedulePC = 22; break;
              }
              ;
```

```
            }

            case 23: /* fire */
            { schedulePC = 24; action_foo();
            }

            case 24: /* fire */
            {
              schedulePC = 25;
              action_bar(((IntSort)IntSort.lit(1)),
                         ((IntSort)IntSort.lit(2)));

            }

            case 25: /* conditional test */
            {
              if
              (baz_v10.booleanValue())
              { schedulePC = 26; break;
              }

              else
              { schedulePC = 21; break;
              }
              ;

            }

            case 26: /* fire */
            {
              schedulePC = 27;
              action_bar(k_v6,
                         ((IntSort)IntSort.add(k_v6,
                                               ((IntSort)IntSort.lit(1)))));

            }

            case 27: /* assignment */
            {
              baz_v10 = ((BoolSort)BoolSort.not(baz_v10));
              schedulePC = 21;
              break;

            }

            case 22: /* schedule ended */
            { break scheduleLoop;
            }
            default:
              throw new Error("NDR␣program␣jumped␣to␣invalid␣PC");};

          }
          ;

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"NDR01"});
}

}
```

Appendix C

Banking Example

C.1 Automaton with NDR Annotations

```
% Automaton implements the B Banking example in Garland and Lynch's
% "Using I/O Automata for Developing Distributed Systems"

% modified by mjt not to use fully qualified state variable references
% in the schedule

% Implementation of null possibility
type OpRec = tuple of  loc: Int,
                       seqno: Int,
                       amount : Int,
                       reported: Bool
type BalRec = tuple of loc: Int,
                       value: Int

% Defines sums over a set
uses NonDet
uses ChoiceSet (OpRec)
uses ChoiceSet (BalRec)

% Sorry no actual parameters to automaton

% This automaton is BankA and Env manually composed

automaton Banking03
signature
  output
    OK(i: Int, x : OpRec),
    reportBalance(n: Int, i: Int)
  internal
    doBalance(i: Int, tempChosenOps : Set[OpRec], amount : Int),
% These are actually inputs to bank
    requestDeposit(n: Int, i: Int) where n > 0,
    requestWithdrawal(n: Int, i: Int) where n > 0,
    requestBalance(i: Int)


  states
```

```
    ops: Set[OpRec] := {},
    pending_ops : Set[OpRec] := {},
    reported_ops : Set[OpRec] := {},
    pending_bals : Set[BalRec] := {},
    done_bals : Set[BalRec] := {},
    bals : Set[BalRec] := {},
    lastSeqno: Array[Int, Int] := constant(0),
    chosenOps: Set[OpRec] := {},

%Environment
    active: Array[Int, Bool] := constant(false),
    % Auxiliary information to help Daikon
    actives : Set[Int] := {},
    not_actives : Set[Int] := {}
%Dummy variables

  transitions
    internal requestDeposit(n, i)
    pre
      % Env
      n > 0 ∧ ¬active[i]
    eff
      lastSeqno[i] := lastSeqno[i] + 1;
      ops := insert([i, lastSeqno[i], n, false], ops);
      pending_ops := insert ([i, lastSeqno[i], n, false], pending_ops);
      % Env
      active[i] := true;
      actives := insert (i, actives);
      not_actives := delete (i, not_actives)



    internal requestWithdrawal(n, i)
    pre
      % Env
      n > 0 ∧ ¬active[i]
    eff
      lastSeqno[i] := lastSeqno[i] + 1;
      ops := insert([i, lastSeqno[i], −n, false], ops);
      pending_ops := insert ([i, lastSeqno[i], −n, false], pending_ops);
      % Env
      active[i] := true;
      actives := insert (i, actives);
      not_actives := delete (i, not_actives)

    internal requestBalance(i)
    pre
      % Env
      ¬active[i]
    eff
      pending_bals := insert ([i, 0], pending_bals);
      bals := pending_bals ∪ done_bals;
      %Env
      active[i] := true;
      actives := insert (i, actives);
      not_actives := delete (i, not_actives)

    output OK(i, x)
      %X isn't a real parameter, it just helps avoid choose
    pre
      x ∈ ops ∧ x.loc = i ∧ ¬x.reported
    eff
      ops := insert(set_reported(x, true), delete(x, ops));
```

```
      pending_ops := delete (x, pending_ops);
      reported_ops := insert (set_reported(x, true), reported_ops);
      % Env
      active[i] := false;
      not_actives := insert (i, not_actives);
      actives := delete (i, actives)


    output reportBalance(n, i)
    pre
      [i, n] ∈ done_bals
    eff
      done_bals := delete([i, n], done_bals);
      bals := pending_bals ∪ done_bals;
      % Env
      active[i] := false;
      not_actives := insert (i, not_actives);
      actives := delete (i, actives)



    internal doBalance(i, tempChosenOps, amount)
      pre
        [i, 0] ∈ pending_bals
      eff
        chosenOps := tempChosenOps;
        pending_bals := delete([i, 0], pending_bals);
        done_bals := insert([i, amount], done_bals);
        bals := pending_bals ∪ done_bals


% Now scheduling blocks to test the automaton

schedule
states
  numLocations : Int,
  location : Int,
  actionChosen : Int,
  maxAmount : Int,
  op : OpRec,
  tempOps : Set[OpRec] := {},
  tempOps2 : Set[OpRec] := {},
  loopBreak : Bool := false,

  tempBals : Set[BalRec] := {},
  bal : BalRec,
  amount : Int

do
  numLocations := 20;
  maxAmount := 1000;

  while (true) do
    % We'll pick a random location now
    location := randomInt (0, numLocations - 1);
    actionChosen := randomInt (0, 10);

    if (actionChosen ≥ 0 ∧ actionChosen ≤ 2) then
      % Do a deposit.  But must be sure we're not active at this location
      if ¬active[location] then
        fire internal requestDeposit(randomInt (1, maxAmount), location)
      fi
    fi;
```

```
    if (actionChosen ≥ 3 ∧ actionChosen ≤ 4) then
      if ¬active[location] then
        fire internal requestWithdrawal (randomInt (1, maxAmount), location)
      fi
    fi;
    if (actionChosen = 5) then
      if ¬active[location] then
        fire internal requestBalance (location)
      fi
    fi;
    if (actionChosen ≥ 6 ∧ actionChosen ≤ 8) then
      tempOps := pending_ops;
      loopBreak := false;
      while (¬isEmpty(tempOps) ∧ ¬loopBreak) do
        op := chooseRandom (ops);
        tempOps := delete (op, tempOps);
        if (¬op.reported) then
          loopBreak := true;
          fire output OK (op.loc, op)
        fi
      od
    fi;
    if (actionChosen = 9) then
      tempBals := done_bals;
      loopBreak := false;
      if (¬isEmpty(tempBals)) then
        bal := chooseRandom (tempBals);
        tempBals := delete (bal, tempBals);
        fire output reportBalance (bal.value, bal.loc)
      fi
    fi;
    if (actionChosen = 10) then
      % Find a null balance
      tempBals := pending_bals;
      loopBreak := false;
      bal := [10, 10];
      if (¬isEmpty(tempBals)) then
        bal := chooseRandom (tempBals);
        tempBals := delete (bal, tempBals);

        % There is a null bal to do balance for
        loopBreak := false;
        tempOps := ops;
        tempOps2 := {};
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps);
          tempOps := delete (op, tempOps);
          if (op.loc = bal.loc) then
            tempOps2 := insert (op, tempOps2)
          fi
        od;
        tempOps := tempOps2;
        amount := 0;
        while (¬isEmpty(tempOps)) do
          op := chooseRandom(tempOps);
          tempOps := delete (op, tempOps);
          amount := amount + op.amount
        od;
        fire internal doBalance (bal.loc, tempOps2, amount)
      fi
    fi
  od
od
```

C.2 Generated Java Code

```
import java.io.*; import ioa.runtime.adt.*; import ioa.runtime.*;

public class Banking03 extends ioa.runtime.Automaton {
  public SetSort ops_v9 = ((SetSort)SetSort.empty());
  public SetSort pending_ops_v10 = ((SetSort)SetSort.empty());
  public SetSort reported_ops_v11 = ((SetSort)SetSort.empty());
  public SetSort pending_bals_v12 = ((SetSort)SetSort.empty());
  public SetSort done_bals_v13 = ((SetSort)SetSort.empty());
  public SetSort bals_v14 = ((SetSort)SetSort.empty());
  public ArraySort lastSeqno_v15 =
    ((ArraySort)ArraySort.constant(((IntSort)IntSort.lit(0))));
  public SetSort chosenOps_v16 = ((SetSort)SetSort.empty());
  public ArraySort active_v17 =
    ((ArraySort)ArraySort.constant(((BoolSort)BoolSort.False())));
  public SetSort actives_v18 = ((SetSort)SetSort.empty());
  public SetSort not_actives_v19 = ((SetSort)SetSort.empty());
  public IntSort numLocations_v20 =
    (IntSort)IntSort.construct(new Parameterization());
  public IntSort location_v21 =
    (IntSort)IntSort.construct(new Parameterization());
  public IntSort actionChosen_v22 =
    (IntSort)IntSort.construct(new Parameterization());
  public IntSort maxAmount_v23 =
    (IntSort)IntSort.construct(new Parameterization());
  public TupleSort op_v24 =
    (TupleSort)TupleSort.construct(new
                                   Parameterization(new Object[]
                                       {new Object[]
                                          {"loc",
                                           "seqno",
                                           "amount",
                                           "reported"},
                                        new Object[]
                                          {ioa.runtime.adt.IntSort.class,
                                           ioa.runtime.adt.IntSort.class,
                                           ioa.runtime.adt.IntSort.class,
                                           ioa.runtime.adt.BoolSort.class},
                                        new Object[]
                                          {new
                                             Parameterization(),
                                           new
                                             Parameterization(),
                                           new
                                             Parameterization(),
                                           new
                                             Parameterization()}}));
  public SetSort tempOps_v25 = ((SetSort)SetSort.empty());
  public SetSort tempOps2_v26 = ((SetSort)SetSort.empty());
  public BoolSort loopBreak_v27 = ((BoolSort)BoolSort.False());
  public SetSort tempBals_v28 = ((SetSort)SetSort.empty());
  public TupleSort bal_v29 =
    (TupleSort)TupleSort.construct(new
                        Parameterization(new Object[]
                                         {new Object[]
                                            {"loc",
                                             "value"},
                                          new Object[]
                                            {ioa.runtime.adt.IntSort.class,
                                             ioa.runtime.adt.IntSort.class},
                                          new Object[]
                                            {new
                                               Parameterization(),
                                             new
                                               Parameterization()}}));
  public IntSort amount_v8 =
    (IntSort)IntSort.construct(new Parameterization());
   int schedulePC = 0;
```

```
public void action_requestDeposit( IntSort n_v6,  IntSort i_v4)
  {
    enteredTransition("internal␣requestDeposit(" + n_v6 +
    ",␣" +i_v4 +")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)(BoolSort.lit(((BoolSort)IntSort.gt(n_v6,
      ((IntSort)IntSort.lit(0)))).booleanValue() &&
      ((BoolSort)BoolSort.not(((BoolSort)ArraySort.elementAt(active_v17,
        i_v4)))).booleanValue()))))).booleanValue());
    lastSeqno_v15 =
      (ArraySort.assign(lastSeqno_v15,
     i_v4,
     ((IntSort)IntSort.add(((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                                         i_v4)),
                           ((IntSort)IntSort.lit(1))))));
    ops_v9 =
      ((SetSort)SetSort.insert(((TupleSort)
           TupleSort.make("loc%seqno%amount%reported%",
                          new Object[]
                            {i_v4,
                             ((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                         i_v4)),
                             n_v6,
                             ((BoolSort)BoolSort.False())})),
                             ops_v9));
    pending_ops_v10 =
      ((SetSort)SetSort.insert(((TupleSort)
           TupleSort.make("loc%seqno%amount%reported%",
                          new Object[]
                            {i_v4,
                             ((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                         i_v4)),
                             n_v6,
                             ((BoolSort)BoolSort.False())})),
                             pending_ops_v10));
    active_v17 =
      (ArraySort.assign(active_v17, i_v4, ((BoolSort)BoolSort.True())));
    actives_v18 = ((SetSort)SetSort.insert(i_v4, actives_v18));
    not_actives_v19 = ((SetSort)SetSort.delete(i_v4, not_actives_v19));
    exitedTransition("internal␣requestDeposit(" + n_v6 +
      ",␣" +i_v4 +")␣in␣automaton␣Banking03");

}

public void action_requestWithdrawal( IntSort n_v6,  IntSort i_v4)
  {
    enteredTransition("internal␣requestWithdrawal(" + n_v6 + ",␣"
      +i_v4 +")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)(BoolSort.lit(((BoolSort)
           IntSort.gt(n_v6,
           ((IntSort)IntSort.lit(0)))).booleanValue() &&
     ((BoolSort)BoolSort.not(((BoolSort)ArraySort.elementAt(active_v17,
             i_v4)))).booleanValue()))))).booleanValue());
    lastSeqno_v15 =
      (ArraySort.assign(lastSeqno_v15,
        i_v4,
        ((IntSort)IntSort.add(((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                                            i_v4)),
                              ((IntSort)IntSort.lit(1))))));
    ops_v9 =
      ((SetSort)SetSort.insert(((TupleSort)
              TupleSort.make("loc%seqno%amount%reported%",
                             new Object[]
                               {i_v4,
                                ((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                            i_v4)),
                                ((IntSort)IntSort.neg(n_v6)),
                                ((BoolSort)BoolSort.False())})),
                                 ops_v9));
    pending_ops_v10 =
      ((SetSort)SetSort.insert(((TupleSort)
```

```
                TupleSort.make("loc%seqno%amount%reported%",
                               new Object[]
                                 {i_v4,
                                  ((IntSort)ArraySort.elementAt(lastSeqno_v15,
                                                    i_v4)),
                                  ((IntSort)IntSort.neg(n_v6)),
                                  ((BoolSort)BoolSort.False())})),
                                   pending_ops_v10));
    active_v17 =
      (ArraySort.assign(active_v17, i_v4, ((BoolSort)BoolSort.True())));
    actives_v18 = ((SetSort)SetSort.insert(i_v4, actives_v18));
    not_actives_v19 = ((SetSort)SetSort.delete(i_v4, not_actives_v19));
    exitedTransition("internal␣requestWithdrawal(" + n_v6 +
      ",␣" +i_v4 +")␣in␣automaton␣Banking03");

}

public void action_requestBalance( IntSort i_v4)
  {
    enteredTransition("internal␣requestBalance(" + i_v4 +
          ")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)BoolSort.not(((BoolSort)
          ArraySort.elementAt(active_v17,
                              i_v4))))).booleanValue());
    pending_bals_v12 =
      ((SetSort)SetSort.insert(((TupleSort)TupleSort.make("loc%value%",
                                          new Object[]
                                            {i_v4,
                                             ((IntSort)IntSort.lit(0))})),
                               pending_bals_v12));
    bals_v14 = ((SetSort)SetSort.union(pending_bals_v12, done_bals_v13));
    active_v17 =
      (ArraySort.assign(active_v17, i_v4, ((BoolSort)BoolSort.True())));
    actives_v18 = ((SetSort)SetSort.insert(i_v4, actives_v18));
    not_actives_v19 = ((SetSort)SetSort.delete(i_v4, not_actives_v19));
    exitedTransition("internal␣requestBalance(" + i_v4
         +")␣in␣automaton␣Banking03");

}

public void action_OK( IntSort i_v4,  TupleSort x_v5)
  {
    enteredTransition("output␣OK(" + i_v4 + ",␣" +x_v5
           +")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)(BoolSort.lit(((BoolSort)
          (BoolSort.lit(((BoolSort)SetSort.isIn(x_v5,
         ops_v9)).booleanValue() && ((BoolSort)IntSort.equals(((IntSort)
              TupleSort.lookupField("loc",
         x_v5)),
         i_v4)).booleanValue())))).booleanValue() && ((BoolSort)
                  BoolSort.not(((BoolSort)TupleSort.lookupField("reported",
         x_v5)))).booleanValue()))))).booleanValue());
    ops_v9 =
      ((SetSort)SetSort.insert(((TupleSort)
           TupleSort.setField("reported",
                                          x_v5,
                                          ((BoolSort)BoolSort.True()))),
                             ((SetSort)SetSort.delete(x_v5, ops_v9))));
    pending_ops_v10 = ((SetSort)SetSort.delete(x_v5, pending_ops_v10));
    reported_ops_v11 =
      ((SetSort)SetSort.insert(((TupleSort)
          TupleSort.setField("reported",
                             x_v5,
                             ((BoolSort)BoolSort.True()))),
                              reported_ops_v11));
    active_v17 =
      (ArraySort.assign(active_v17, i_v4, ((BoolSort)BoolSort.False())));
    not_actives_v19 = ((SetSort)SetSort.insert(i_v4, not_actives_v19));
    actives_v18 = ((SetSort)SetSort.delete(i_v4, actives_v18));
    exitedTransition("output␣OK(" + i_v4 + ",␣" +x_v5 +
```

```
      ")␣in␣automaton␣Banking03");

}

public void action_reportBalance( IntSort n_v6,  IntSort i_v4)
  {
    enteredTransition("output␣reportBalance(" + n_v6 + ",␣" +
       i_v4 +")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)SetSort.isIn(((TupleSort)
        TupleSort.make("loc%value%",
                                                    new
                                                      Object[]
                                                      {i_v4,
                                                    n_v6})),
                            done_bals_v13))).booleanValue());
    done_bals_v13 =
      ((SetSort)SetSort.delete(((TupleSort)TupleSort.make("loc%value%",
                                                          new Object[]
                                                            {i_v4,
                                                             n_v6})),
                               done_bals_v13));
    bals_v14 = ((SetSort)SetSort.union(pending_bals_v12, done_bals_v13));
    active_v17 =
      (ArraySort.assign(active_v17, i_v4, ((BoolSort)BoolSort.False())));
    not_actives_v19 = ((SetSort)SetSort.insert(i_v4, not_actives_v19));
    actives_v18 = ((SetSort)SetSort.delete(i_v4, actives_v18));
    exitedTransition("output␣reportBalance(" + n_v6 + ",␣"
       +i_v4 +")␣in␣automaton␣Banking03");

}

public void
  action_doBalance( IntSort i_v4,
                    SetSort tempChosenOps_v7,
                    IntSort amount_v8)
  {
    enteredTransition("internal␣doBalance(" + i_v4 + ",␣" +
      tempChosenOps_v7 + ",␣" +amount_v8 +")␣in␣automaton␣Banking03");
    assertPrecondition((((BoolSort)SetSort.isIn(((TupleSort)
                 TupleSort.make("loc%value%",
                                             new
                                               Object[]
                                               {i_v4,
                                             ((IntSort)IntSort.lit(0))})),
                  pending_bals_v12))).booleanValue());
    chosenOps_v16 = tempChosenOps_v7;
    pending_bals_v12 =
      ((SetSort)SetSort.delete(((TupleSort)TupleSort.make("loc%value%",
                                       new Object[]
                                         {i_v4,
                                          ((IntSort)IntSort.lit(0))})),
                               pending_bals_v12));
    done_bals_v13 =
      ((SetSort)SetSort.insert(((TupleSort)TupleSort.make("loc%value%",
                                                          new Object[]
                                                            {i_v4,
                                                             amount_v8})),
                               done_bals_v13));
    bals_v14 = ((SetSort)SetSort.union(pending_bals_v12, done_bals_v13));
    exitedTransition("internal␣doBalance(" + i_v4 + ",␣" +tempChosenOps_v7
    + ",␣" +amount_v8 +")␣in␣automaton␣Banking03");

}

protected void schedule()
  {
    scheduleLoop: while ( true )
      {
        switch ( schedulePC ) {
            case 0: /* assignment */
```

```
{
  numLocations_v20 = ((IntSort)IntSort.lit(20));
  schedulePC = 2;
  break;

}

case 2: /* assignment */
{
  maxAmount_v23 = ((IntSort)IntSort.lit(1000));
  schedulePC = 3;
  break;

}

case 3: /* while loop test */
{
  if
  (((BoolSort)BoolSort.True()).booleanValue())
  { schedulePC = 4; break;
  }

  else
  { schedulePC = 1; break;
  }
  ;

}

case 4: /* assignment */
{
  location_v21 =
    ((IntSort)NonDet.randomInt(((IntSort)IntSort.lit(0)),
            ((IntSort)IntSort.sub(numLocations_v20,
                                  ((IntSort)IntSort.lit(1))))));
  schedulePC = 5;
  break;

}

case 5: /* assignment */
{
  actionChosen_v22 =
    ((IntSort)NonDet.randomInt(((IntSort)IntSort.lit(0)),
                               ((IntSort)IntSort.lit(10))));
  schedulePC = 6;
  break;

}

case 6: /* conditional test */
{
  if
  (((BoolSort)(BoolSort.lit(((BoolSort)
   IntSort.gte(actionChosen_v22,
   ((IntSort)IntSort.lit(0)))).booleanValue() &&
   ((BoolSort)IntSort.lte(actionChosen_v22,
    ((IntSort)IntSort.lit(2)))).booleanValue()))).booleanValue())
  { schedulePC = 8; break;
  }

  else
  { schedulePC = 7; break;
  }
  ;

}

case 8: /* conditional test */
{
```

```
  if
  (((BoolSort)BoolSort.not(((BoolSort)
    ArraySort.elementAt(active_v17,
                location_v21)))).booleanValue())
  { schedulePC = 9; break;
  }

  else
  { schedulePC = 7; break;
  }
  ;

}

case 9: /* fire */
{
  schedulePC = 7;
  action_requestDeposit(((IntSort)
   NonDet.randomInt(((IntSort)IntSort.lit(1)),
                                       maxAmount_v23)),
                        location_v21);

}

case 7: /* conditional test */
{
  if
  (((BoolSort)(BoolSort.lit(((BoolSort)IntSort.gte(actionChosen_v22,
      ((IntSort)IntSort.lit(3)))).booleanValue() &&
      ((BoolSort)IntSort.lte(actionChosen_v22,
    ((IntSort)IntSort.lit(4)))).booleanValue()))).booleanValue())
  { schedulePC = 11; break;
  }

  else
  { schedulePC = 10; break;
  }
  ;

}

case 11: /* conditional test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
    ArraySort.elementAt(active_v17,
                             location_v21)))).booleanValue())
  { schedulePC = 12; break;
  }

  else
  { schedulePC = 10; break;
  }
  ;

}

case 12: /* fire */
{
  schedulePC = 10;
  action_requestWithdrawal(((IntSort)
        NonDet.randomInt(((IntSort)IntSort.lit(1)),
                                         maxAmount_v23)),
                           location_v21);

}

case 10: /* conditional test */
{
  if
```

```
  (((BoolSort)IntSort.equals(actionChosen_v22,
                             ((IntSort)
                             IntSort.lit(5)))).booleanValue())
  { schedulePC = 14; break;
  }

  else
  { schedulePC = 13; break;
  }
  ;

}

case 14: /* conditional test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
       ArraySort.elementAt(active_v17,
                           location_v21)))).booleanValue())
  { schedulePC = 15; break;
  }

  else
  { schedulePC = 13; break;
  }
  ;

}

case 15: /* fire */
{ schedulePC = 13; action_requestBalance(location_v21);
}

case 13: /* conditional test */
{
  if
  (((BoolSort)(BoolSort.lit(((BoolSort)
        IntSort.gte(actionChosen_v22,
          ((IntSort)IntSort.lit(6)))).booleanValue() &&
          ((BoolSort)IntSort.lte(actionChosen_v22,
     ((IntSort)IntSort.lit(8)))).booleanValue()))).booleanValue())
  { schedulePC = 17; break;
  }

  else
  { schedulePC = 16; break;
  }
  ;

}

case 17: /* assignment */
{ tempOps_v25 = pending_ops_v10; schedulePC = 18; break;
}

case 18: /* assignment */
{
  loopBreak_v27 = ((BoolSort)BoolSort.False());
  schedulePC = 19;
  break;

}

case 19: /* while loop test */
{
  if
  (((BoolSort)(BoolSort.lit(((BoolSort)BoolSort.not(((BoolSort)
        SetSort.isEmpty(tempOps_v25)))).booleanValue() &&
        ((BoolSort)BoolSort.not(loopBreak_v27)).
        booleanValue()))).booleanValue())
```

```
  { schedulePC = 20; break;
  }

  else
  { schedulePC = 16; break;
  }
  ;

}

case 20: /* assignment */
{
  op_v24 = ((TupleSort)SetSort.chooseRandom(ops_v9));
  schedulePC = 21;
  break;

}

case 21: /* assignment */
{
  tempOps_v25 =
    ((SetSort)SetSort.delete(op_v24, tempOps_v25));
  schedulePC = 22;
  break;

}

case 22: /* conditional test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
     TupleSort.lookupField("reported",
                           op_v24)))).booleanValue())
  { schedulePC = 23; break;
  }

  else
  { schedulePC = 19; break;
  }
  ;

}

case 23: /* assignment */
{
  loopBreak_v27 = ((BoolSort)BoolSort.True());
  schedulePC = 24;
  break;

}

case 24: /* fire */
{
  schedulePC = 19;
  action_OK(((IntSort)TupleSort.lookupField("loc", op_v24)),
            op_v24);

}

case 16: /* conditional test */
{
  if
  (((BoolSort)IntSort.equals(actionChosen_v22,
                             ((IntSort)
                             IntSort.lit(9)))).booleanValue())
  { schedulePC = 26; break;
  }

  else
  { schedulePC = 25; break;
```

```
  }
  ;

}

case 26: /* assignment */
{ tempBals_v28 = done_bals_v13; schedulePC = 27; break;
}

case 27: /* assignment */
{
  loopBreak_v27 = ((BoolSort)BoolSort.False());
  schedulePC = 28;
  break;

}

case 28: /* conditional test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
               SetSort.isEmpty(tempBals_v28)))).booleanValue())
  { schedulePC = 29; break;
  }

  else
  { schedulePC = 25; break;
  }
  ;

}

case 29: /* assignment */
{
  bal_v29 = ((TupleSort)SetSort.chooseRandom(tempBals_v28));
  schedulePC = 30;
  break;

}

case 30: /* assignment */
{
  tempBals_v28 =
    ((SetSort)SetSort.delete(bal_v29, tempBals_v28));
  schedulePC = 31;
  break;

}

case 31: /* fire */
{
  schedulePC = 25;
  action_reportBalance(((IntSort)TupleSort.lookupField("value",
                                                       bal_v29)),
                       ((IntSort)TupleSort.lookupField("loc",
                                                       bal_v29)));

}

case 25: /* conditional test */
{
  if
  (((BoolSort)IntSort.equals(actionChosen_v22,
                             ((IntSort)IntSort.lit(10)))).
                             booleanValue())
  { schedulePC = 32; break;
  }

  else
  { schedulePC = 3; break;
```

```
  }
  ;

}

case 32: /* assignment */
{ tempBals_v28 = pending_bals_v12; schedulePC = 33; break;
}

case 33: /* assignment */
{
  loopBreak_v27 = ((BoolSort)BoolSort.False());
  schedulePC = 34;
  break;

}

case 34: /* assignment */
{
  bal_v29 =
    ((TupleSort)TupleSort.make("loc%value%",
                               new Object[]
                                 {((IntSort)IntSort.lit(10)),
                                  ((IntSort)IntSort.lit(10))}));
  schedulePC = 35;
  break;

}

case 35: /* conditional test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
      SetSort.isEmpty(tempBals_v28)))).booleanValue())
  { schedulePC = 36; break;
  }

  else
  { schedulePC = 3; break;
  }
  ;

}

case 36: /* assignment */
{
  bal_v29 = ((TupleSort)SetSort.chooseRandom(tempBals_v28));
  schedulePC = 37;
  break;

}

case 37: /* assignment */
{
  tempBals_v28 =
    ((SetSort)SetSort.delete(bal_v29, tempBals_v28));
  schedulePC = 38;
  break;

}

case 38: /* assignment */
{
  loopBreak_v27 = ((BoolSort)BoolSort.False());
  schedulePC = 39;
  break;

}

case 39: /* assignment */
```

```
{ tempOps_v25 = ops_v9; schedulePC = 40; break;
}

case 40: /* assignment */
{
  tempOps2_v26 = ((SetSort)SetSort.empty());
  schedulePC = 41;
  break;

}

case 41: /* while loop test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
      SetSort.isEmpty(tempOps_v25)))).booleanValue())
  { schedulePC = 43; break;
  }

  else
  { schedulePC = 42; break;
  }
  ;

}

case 43: /* assignment */
{
  op_v24 = ((TupleSort)SetSort.chooseRandom(tempOps_v25));
  schedulePC = 44;
  break;

}

case 44: /* assignment */
{
  tempOps_v25 =
    ((SetSort)SetSort.delete(op_v24, tempOps_v25));
  schedulePC = 45;
  break;

}

case 45: /* conditional test */
{
  if
  (((BoolSort)IntSort.equals(((IntSort)
                              TupleSort.lookupField("loc",
                                                    op_v24)),
                              ((IntSort)
                              TupleSort.lookupField("loc",
                                   bal_v29)))).booleanValue())
  { schedulePC = 46; break;
  }

  else
  { schedulePC = 41; break;
  }
  ;

}

case 46: /* assignment */
{
  tempOps2_v26 =
    ((SetSort)SetSort.insert(op_v24, tempOps2_v26));
  schedulePC = 41;
  break;

}
```

```
case 42: /* assignment */
{ tempOps_v25 = tempOps2_v26; schedulePC = 47; break;
}

case 47: /* assignment */
{
  amount_v8 = ((IntSort)IntSort.lit(0));
  schedulePC = 48;
  break;

}

case 48: /* while loop test */
{
  if
  (((BoolSort)BoolSort.not(((BoolSort)
      SetSort.isEmpty(tempOps_v25)))).booleanValue())
  { schedulePC = 50; break;
  }

  else
  { schedulePC = 49; break;
  }
  ;

}

case 50: /* assignment */
{
  op_v24 = ((TupleSort)SetSort.chooseRandom(tempOps_v25));
  schedulePC = 51;
  break;

}

case 51: /* assignment */
{
  tempOps_v25 =
    ((SetSort)SetSort.delete(op_v24, tempOps_v25));
  schedulePC = 52;
  break;

}

case 52: /* assignment */
{
  amount_v8 =
    ((IntSort)IntSort.add(amount_v8,
                ((IntSort)TupleSort.lookupField("amount",
                                                op_v24))));
  schedulePC = 48;
  break;

}

case 49: /* fire */
{
  schedulePC = 3;
  action_doBalance(((IntSort)TupleSort.lookupField("loc",
                                                   bal_v29)),
                   tempOps2_v26,
                   amount_v8);

}

case 1: /* schedule ended */
{ break scheduleLoop;
}
default:
```

```
                throw new Error("NDR program jumped to invalid PC");};

            }
            ;

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"Banking03"});
}

}
```

C.3 Trace

```
[[[[ Begin initialization [[[[
%%%% Modified state variables:
     actionChosen --> 87
     active --> (ArraySort (ConstantValue false))
     actives --> ()
     amount --> 87
     bal --> [87, 87]
     bals --> ()
     chosenOps --> ()
     done_bals --> ()
     lastSeqno --> (ArraySort (ConstantValue 0))
     location --> 87
     loopBreak --> false
     maxAmount --> 87
     not_actives --> ()
     numLocations --> 87
     op --> [87, 87, 87, true]
     ops --> ()
     pending_bals --> ()
     pending_ops --> ()
     reported_ops --> ()
     tempBals --> ()
     tempOps --> ()
     tempOps2 --> ()
]]]] End initialization ]]]]
[[[[ transition: internal requestDeposit(14, 11) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 2
     active --> (ArraySort (ConstantValue false) (11 true))
     actives --> (11)
     amount --> 87
     bal --> [87, 87]
     bals --> ()
     chosenOps --> ()
     done_bals --> ()
     lastSeqno --> (ArraySort (ConstantValue 0) (11 1))
     location --> 11
     loopBreak --> false
     maxAmount --> 1000
     not_actives --> ()
     numLocations --> 20
     op --> [87, 87, 87, true]
     ops --> ([11, 1, 14, false])
     pending_bals --> ()
     pending_ops --> ([11, 1, 14, false])
     reported_ops --> ()
     tempBals --> ()
     tempOps --> ()
     tempOps2 --> ()
]]]]
```

```
[[[[ transition: internal requestWithdrawal(739, 3) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 3
     active --> (ArraySort (ConstantValue false) (11 true) (3 true))
     actives --> (11 3)
     lastSeqno --> (ArraySort (ConstantValue 0) (11 1) (3 1))
     location --> 3
     not_actives --> ()
     ops --> ([11, 1, 14, false] [3, 1, -739, false])
     pending_ops --> ([11, 1, 14, false] [3, 1, -739, false])
]]]]
[[[[ transition: internal requestWithdrawal(411, 7) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 4
     active --> (ArraySort (ConstantValue false) (11 true) (3 true) (7 true))
     actives --> (11 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (11 1) (3 1) (7 1))
     location --> 7
     not_actives --> ()
     ops --> ([11, 1, 14, false] [3, 1, -739, false] [7, 1, -411, false])
     pending_ops --> ([11, 1, 14, false] [3, 1, -739, false] [7, 1, -411,
false])
]]]]
[[[[ transition: internal requestDeposit(844, 10) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 1
     active --> (ArraySort (ConstantValue false) (10 true) (11 true) (3 true) (7
true))
     actives --> (10 11 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (10 1) (11 1) (3 1) (7 1))
     location --> 10
     not_actives --> ()
     ops --> ([10, 1, 844, false] [11, 1, 14, false] [3, 1, -739, false] [7, 1,
-411, false])
     pending_ops --> ([10, 1, 844, false] [11, 1, 14, false] [3, 1, -739, false]
[7, 1, -411, false])
]]]]
[[[[ transition: internal requestDeposit(246, 0) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 1
     active --> (ArraySort (ConstantValue false) (0 true) (10 true) (11 true) (3
true) (7 true))
     actives --> (0 10 11 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (0 1) (10 1) (11 1) (3 1) (7 1))
     location --> 0
     not_actives --> ()
     ops --> ([0, 1, 246, false] [10, 1, 844, false] [11, 1, 14, false] [3, 1,
-739, false] [7, 1, -411, false])
     pending_ops --> ([0, 1, 246, false] [10, 1, 844, false] [11, 1, 14, false]
[3, 1, -739, false] [7, 1, -411, false])
]]]]
[[[[ transition: output OK(11, [11, 1, 14, false]) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 6
     active --> (ArraySort (ConstantValue false) (0 true) (10 true) (11 false)
(3 true) (7 true))
     actives --> (0 10 3 7)
     location --> 9
     loopBreak --> true
     not_actives --> (11)
     op --> [11, 1, 14, false]
     ops --> ([0, 1, 246, false] [10, 1, 844, false] [11, 1, 14, true] [3, 1,
-739, false] [7, 1, -411, false])
     pending_ops --> ([0, 1, 246, false] [10, 1, 844, false] [3, 1, -739, false]
[7, 1, -411, false])
     reported_ops --> ([11, 1, 14, true])
     tempOps --> ([0, 1, 246, false] [10, 1, 844, false] [3, 1, -739, false] [7,
1, -411, false])
```

```
]]]]
[[[[ transition: output OK(10, [10, 1, 844, false]) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 7
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(3 true) (7 true))
     actives --> (0 3 7)
     bal --> [10, 10]
     location --> 6
     not_actives --> (10 11)
     op --> [10, 1, 844, false]
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [3, 1,
-739, false] [7, 1, -411, false])
     pending_ops --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411,
false])
     reported_ops --> ([10, 1, 844, true] [11, 1, 14, true])
     tempBals --> ()
     tempOps --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411, false])
]]]]
[[[[ transition: internal requestDeposit(811, 8) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 1
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(3 true) (7 true) (8 true))
     actives --> (0 3 7 8)
     lastSeqno --> (ArraySort (ConstantValue 0) (0 1) (10 1) (11 1) (3 1) (7 1)
(8 1))
     location --> 8
     not_actives --> (10 11)
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [3, 1,
-739, false] [7, 1, -411, false] [8, 1, 811, false])
     pending_ops --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411, false]
[8, 1, 811, false])
]]]]
[[[[ transition: output OK(8, [8, 1, 811, false]) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 6
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(3 true) (7 true) (8 false))
     actives --> (0 3 7)
     location --> 17
     not_actives --> (10 11 8)
     op --> [8, 1, 811, false]
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [3, 1,
-739, false] [7, 1, -411, false] [8, 1, 811, true])
     pending_ops --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411,
false])
     reported_ops --> ([10, 1, 844, true] [11, 1, 14, true] [8, 1, 811, true])
     tempOps --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411, false])
]]]]
[[[[ transition: internal requestWithdrawal(930, 17) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 3
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(17 true) (3 true) (7 true) (8 false))
     actives --> (0 17 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (0 1) (10 1) (11 1) (17 1) (3 1)
(7 1) (8 1))
     location --> 17
     not_actives --> (10 11 8)
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [17, 1,
-930, false] [3, 1, -739, false] [7, 1, -411, false] [8, 1, 811, true])
     pending_ops --> ([0, 1, 246, false] [17, 1, -930, false] [3, 1, -739,
false] [7, 1, -411, false])
]]]]
[[[[ transition: output OK(17, [17, 1, -930, false]) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 8
```

```
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(17 false) (3 true) (7 true) (8 false))
     actives --> (0 3 7)
     location --> 2
     not_actives --> (10 11 17 8)
     op --> [17, 1, -930, false]
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [17, 1,
-930, true] [3, 1, -739, false] [7, 1, -411, false] [8, 1, 811, true])
     pending_ops --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411,
false])
     reported_ops --> ([10, 1, 844, true] [11, 1, 14, true] [17, 1, -930, true]
[8, 1, 811, true])
     tempOps --> ([0, 1, 246, false] [3, 1, -739, false] [7, 1, -411, false])
]]]]
[[[[ transition: internal requestDeposit(562, 18) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 0
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(17 false) (18 true) (3 true) (7 true) (8 false))
     actives --> (0 18 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (0 1) (10 1) (11 1) (17 1) (18
1) (3 1) (7 1) (8 1))
     location --> 18
     loopBreak --> false
     not_actives --> (10 11 17 8)
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [17, 1,
-930, true] [18, 1, 562, false] [3, 1, -739, false] [7, 1, -411, false] [8, 1,
811, true])
     pending_ops --> ([0, 1, 246, false] [18, 1, 562, false] [3, 1, -739, false]
[7, 1, -411, false])
     tempBals --> ()
]]]]
[[[[ transition: internal requestDeposit(471, 14) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 2
     active --> (ArraySort (ConstantValue false) (0 true) (10 false) (11 false)
(14 true) (17 false) (18 true) (3 true) (7 true) (8 false))
     actives --> (0 14 18 3 7)
     lastSeqno --> (ArraySort (ConstantValue 0) (0 1) (10 1) (11 1) (14 1) (17
1) (18 1) (3 1) (7 1) (8 1))
     location --> 14
     not_actives --> (10 11 17 8)
     ops --> ([0, 1, 246, false] [10, 1, 844, true] [11, 1, 14, true] [14, 1,
471, false] [17, 1, -930, true] [18, 1, 562, false] [3, 1, -739, false] [7, 1,
-411, false] [8, 1, 811, true])
     pending_ops --> ([0, 1, 246, false] [14, 1, 471, false] [18, 1, 562, false]
[3, 1, -739, false] [7, 1, -411, false])
]]]]
[[[[ transition: output OK(0, [0, 1, 246, false]) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 8
     active --> (ArraySort (ConstantValue false) (0 false) (10 false) (11 false)
(14 true) (17 false) (18 true) (3 true) (7 true) (8 false))
     actives --> (14 18 3 7)
     location --> 11
     loopBreak --> true
     not_actives --> (0 10 11 17 8)
     op --> [0, 1, 246, false]
     ops --> ([0, 1, 246, true] [10, 1, 844, true] [11, 1, 14, true] [14, 1,
471, false] [17, 1, -930, true] [18, 1, 562, false] [3, 1, -739, false] [7, 1,
-411, false] [8, 1, 811, true])
     pending_ops --> ([14, 1, 471, false] [18, 1, 562, false] [3, 1, -739,
false] [7, 1, -411, false])
     reported_ops --> ([0, 1, 246, true] [10, 1, 844, true] [11, 1, 14, true]
[17, 1, -930, true] [8, 1, 811, true])
     tempOps --> ([14, 1, 471, false] [18, 1, 562, false] [3, 1, -739, false]
[7, 1, -411, false])
]]]]
```

```
[[[[ transition: internal requestBalance(6) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 5
     active --> (ArraySort (ConstantValue false) (0 false) (10 false) (11 false)
(14 true) (17 false) (18 true) (3 true) (6 true) (7 true) (8 false))
     actives --> (14 18 3 6 7)
     bals --> ([6, 0])
     location --> 6
     not_actives --> (0 10 11 17 8)
     pending_bals --> ([6, 0])
]]]]
[[[[ transition: internal doBalance(6, (), 0) in automaton Banking03
%%%% Modified state variables:
     actionChosen --> 10
     amount --> 0
     bal --> [6, 0]
     bals --> ([6, 0])
     chosenOps --> ()
     done_bals --> ([6, 0])
     location --> 14
     loopBreak --> false
     op --> [7, 1, -411, false]
     pending_bals --> ()
     tempBals --> ()
     tempOps --> ()
     tempOps2 --> ()
]]]]
```

Bibliography

[1] Mark Baker, Bryan Carpenter, Sung Hoon Ko, and Xinying Li. *mpiJava: A Java interface to MPI.* Submitted to the First UK Workshop on Java for High Performance Network Computing, Europar 1998.

[2] Kent Beck and Erich Gamma. JUnit Testing Framework.
`http://www.junit.org`

[3] Andrej Bogdanov. *Formal verification of simulations between I/O automata.* Master of Engineering thesis, MIT, 2001.
`http://theory.lcs.mit.edu/tds/papers/Bogdanov/thesis.pdf`

[4] Anna E. Chefter. *A Simulator for the IOA Language.* Master of Engineering thesis, MIT, 1998.
`http://theory.lcs.mit.edu/tds/papers/Chefter/thesis.html`

[5] Laura G. Dean. *Improved Simulation of Input/Output Automata.* Master of Engineering thesis, MIT, 2001.
`http://theory.lcs.mit.edu/tds/papers/Dean/thesis.html`

[6] Martin Fowler et al. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[7] Stanislav Funiak. *Model Checking IOA Programs with TLC.* Manuscript, 2001.
`http://theory.lcs.mit.edu/tds/papers/Funiak/report.ps`

[8] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[9] Steven J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming, and Validating Distribued Systems.* Laboratory for Computer Science, MIT, December 2000.
`http://theory.lcs.mit.edu/tds/papers/Garland/ioaManual.ps.gz`

[10] Stephen J. Garland and Nancy A. Lynch. *The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems.* Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, MIT, August 1998 (original version: September 25, 1997).
`http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`

[11] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
`http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps`

[12] Dilsun Kirli Kaynar, Anna Chefter, Laura Dean, Stephen Garland, Nancy Lynch, Toh Ne Win, Antonio Ramírez-Robredo. *The IOA Simulator.* Technical Report, MIT, April 2002.
`http://theory.lcs.mit.edu/~dilsun/Publications/SimManual.ps`

[13] Bill Joy, Guy Steele, James Gosling, Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming.* Budapest, Hungary, 18–22 June 2001.
`http://aspectj.org/documentation/papersAndSlides/ECOOP2001-Overview.pdf`

[15] Barbara Liskov, et al. *CLU Reference Manual,* Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, Cambridge, MA, October 1979.

[16] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, March 1996.

[17] Tim Mackinnon, Steve Freeman, Philip Craig. *Endo-Testing: Unit Testing with Mock Objects.* eXtreme Programming and Flexible Processes in Software Engineering - XP2000.
`http://www.mockobjects.com/misc/mockobjects.pdf`

[18] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[19] Atish Dev Nigam. *Enhancing the IOA Code Generator's Abstract Data Types.* Manuscript, 2001.
`http://theory.lcs.mit.edu/tds/papers/Nigam/report.pdf`

[20] J. Antonio Ramírez-Robredo. *Paired Simulation of I/O Automata.* Master of Engineering thesis, MIT, 2000.
`http://theory.lcs.mit.edu/tds/papers/Ramirez/thesis.html`

[21] Holly Reimers. *Two Translators for the IOA Toolkit.* Advanced Undergraduate Project, MIT, 2000.
`http://theory.lcs.mit.edu/tds/papers/Reimers/final2.ps.gz`

[22] Sun Microsystems. *Javadocs for* `hashCode()`.
`http://java.sun.com/products/jdk/1.1/docs/api/java.lang.Object.html`

[23] Joshua A. Tauber. *Verifiable Code Generation from Abstract I/O Automata Models for Distributed Computing.* PhD thesis proposal, MIT, March 2001.
`http://theory.lcs.mit.edu/~josh/papers/proposal.ps.gz`

[24] Michael J. Tsai *Abstract Data Types for IOA Code Generation.* Manuscript, 2001.
`http://theory.lcs.mit.edu/~mjt/ADTsForIOACodeGeneration.pdf`

[25] Michael J. Tsai. *Comparison of ADTs for the IOA Code Generator and Simulator.* Slides, 2001.
`http://theory.lcs.mit.edu/~mjt/adt-presentation.pdf`

[26] Michael J. Tsai. *Design and Implementation of the IOA GUI, Interface Generator, and NAD Modules.* Manuscript, 2000.
`http://theory.lcs.mit.edu/~mjt/mjtDesignAndImpl.pdf`

[27] Franklyn Turbak and David Gifford with Brian Reistad. *Applied Semantics of Programming Languages.* Course notes for MIT 6.821, draft of August 19, 2001.

[28] Mandana Vaziri, Joshua A. Tauber, and Nancy A. Lynch. *A transformation of I/O automata removing implicit nondeterminism.* Manuscript, 1998.

[29] Mandana Vaziri, Joshua A. Tauber, Michael Tsai, Nancy Lynch. *Compilation of I/O Automata into Executable Code: Removing Nondeterminism.* Manuscript, 1999.

[30] Toh Ne Win. *Implementing Dynamic Sorts for IOA.* Manuscript, 2001.

[31] Toh Ne Win. *Logging in the IOA Toolkit.* Manuscript, 2001.
`http://theory.lcs.mit.edu/~tohn/logging.html`

[32] Toh Ne Win and Gustavo Santos. *A Test for the IOA-Daikon Connection: Banking Examples in IOA.* Manuscript, 2001.
`http://sdg.lcs.mit.edu/~tohn/home/research/papers/banking.ps`