School of Electronics and Computer Science
Faculty of Engineering, Science and Mathematics
University of Southampton

Stephen Tuttlebee

03/05/2011

# An Experimental Language for Concurrency: Components and Synchronisation on Explicit Boundaries

Project supervisor: Dr Pawel Sobocinski
Second examiner: Professor Michael Butler

A project report submitted for the award of Computer Science MEng

# Abstract

Concurrency has long been one of the most challenging areas in programming and in Computer Science in general. In the last few years however, its prominence has increased considerably as Moore’s Law is only sustained by adding more cores to processor chips. Concurrency issues such as safety, deadlock and other liveness issues has encouraged researchers to explore new paradigms and develop languages and libraries that reduce these difficulties.

This project's goals have been similar, involving the design of an experimental language (JavaB) aimed to simplify concurrent application development for software engineers. JavaB's core concepts are underpinned by Sobocinski's ongoing research in process algebra. One benefit of the language from a Software Engineering standpoint is the ability to compose components into 'higher-order' composition components. Such composability allows sophisticated synchronisation behaviour to be encapsulated in a single (composition) component.

In addition to exploration into precise language semantics, the end product of the project is a basic source-to-source translator (that converts JavaB to pure Java), and Java classes that implement JavaB's core synchronisation primitives. The project was an ambitious one. Time constraints meant some planned features were not implemented. Nevertheless, the project provides a sound basis for future work.

# Contents

# Acknowledgements

I am grateful to my supervisor, Dr Pawel Sobocinski, for his encouragement and advice. I appreciated his rapid e-mail replies to my questions, and his willingness to give me time, sometimes at short notice. Without our many discussions, this project probably would not have been possible.

I want to thank Professor Michael Butler for his input also.

I would like to thank my mother for listening to me and giving both pragmatic advice and even attempts at technical advice (which surprisingly to me, did sometimes prove helpful), and also my father for his occasional nuggets of wisdom.

# Statement of Originality

The core ideas behind the JavaB language are based on Sobocinski's research on process algebra. The refinement of the ideas into precise language semantics took place in collaboration with him. Many of the language examples used in this report were also his. I carried out the implementation work for the JavaB language, which included the translation mechanisms and the translator itself. Implementation ideas and problems were discussed with Sobocinski, in an effort to produce solutions. In particular, his suggestion for resolving a deadlock in the NormalWire class was employed.

Yang Jiang's Java grammar was used as a starting point of ANTLR grammars used in translator implementation [**1**]. The majority of code is my own. Where code from external sources has been utilised, this is acknowledged within the source code.

# 1.

# Introduction

*Problem*
Concurrency has long been one of the most challenging areas in programming and Computer Science in general. In the last few years however, its prominence has increased considerably. With Moore's Law no longer translating into performance improvement through the increase of clock speeds, the likes of Intel and AMD are instead adding more cores onto their processor chips in an effort to maintain the performance trend[**2**]. Whilst adding more processing power may be easy enough for chip manufacturers, writing software that utilises it using the existing paradigms, languages and libraries remains a difficult task [**3**] [**4**]. Thus the exploration of new paradigms and enhancement of existing paradigms for concurrency are active research areas. Moreover, there are many efforts in the implementation of new languages and supplementary libraries and tools. These streams of research all generally have the aim of simplifying concurrency for programmers. The different languages, libraries and tools resulting from such research have had various degrees of success.

*Aspirations*
This project introduces an experimental language, JavaB[1], an extension to the Java language, which is hoped will be a feasible language to develop concurrent programs in. JavaB is based on the concepts of components and boundaries. Components can be thought of as somewhat similar to objects in typical Object-Oriented languages (e.g. Java, C++). Components declare boundaries that indicate explicitly the points through which they can synchronise with other components, passing values between them. A vision of the language is that components may be composed into 'higher-order' compositions, which may be treated as components themselves. Primitive synchronisation policies may be encapsulated in a standard set of components. Composing such standard components with other such components to achieve sophisticated synchronisation behaviour encapsulated *in a single, reusable component*, is another vision of the language. JavaB's core concepts are underpinned by Sobocinski's ongoing research in process algebra. Being rooted in such formalisms also gives possible scope for formally verifying the correctness of JavaB programs.

*Project Scope*
This project's scope is limited to early work on this language. Its focus has been three-fold. One focus has been the exploration into the precise semantics (and syntax) of the language itself, which took place in collaboration with Sobocinski. The second, and primary focus, was the design and implementation of a source-to-source translator (JavaB-to-Java). Thirdly, the construction of the above-mentioned synchronisation primitive standard components has been another goal.

[1] The name 'JavaB' derives from the fact that the language extends Java and from the key language concept of a *boundary*.

*Report Structure*

The structure of the report is as follows:

- Chapter 2 reviews background reading undertaken and existing literature in the fields of concurrency and translator implementation.
- Chapters 3 and 4 form the main backbone of the report. Chapter 3 describes the JavaB language. Chapter 4 focuses on the design and implementation of the JavaB translator, including the implementation of the language synchronisation primitives.
- The remaining chapters discuss the testing undertaken, the development process and tools used, and project management.
- The report concludes by summarising and evaluating the project's achievements and also considers future work.

# 2.

# Background Reading and Report of Literature Search

The background research began by gaining familiarity with the concurrency mechanisms and libraries available in Java (monitors, locking, conditions, java.util.concurrent library [**5**]), since the main requirement of the project was to produce a translator that generates Java code which implemented/simulated the semantics of JavaB (this inevitably required Java's concurrency constructs). Books by Goetz[**6**] and Lea [**7**] provided essential reading. The university courses[**8**], [**9**], [**10**] and[**11**] were also helpful.

## 2.1 Concurrency

The field of concurrency is vast, both theoretically and in practice. The following subsections cover only the most important topics.

### 2.1.1 Paradigms (Languages and Libraries)

The two traditional paradigms for thread/process communication are shared-memory and message-passing [**12**]. Examples of languages using a shared-memory model are Java, C and C++ (using pthreads), C# and Python. Shared-memory approaches typically rely on the use of locks, which are used to guard access to shared variables. Even if problems of *safety* ('nothing bad ever happens'; e.g. race conditions, memory visibility problems) are overcome, *liveness* ('something good eventually happens') problems can be introduced instead (e.g. deadlock, livelock, starvation). Locking is not the only strategy though. Non-blocking algorithms which are lock-free also exist. However, writing such algorithms efficiently is difficult and they only exist for the most common data structures [**6**][**13**] [**14**]. For message-passing, recent examples include Erlang, Scala, Axum and Go.

Three surveys of the field highlight several other existing and emerging (and re-emerging) paradigms [**15**] [**16**] [**17**]. Transactional Memory (TM) is one such paradigm [**18**] in which transactions are applied to memory locations rather than database rows/tables. At present, TM is only feasibly implemented in software (STM) [**19**] [**20**][**21**]. New languages such as Clojure[**22**], Fortress[**23**] and Scala[**24**] support STM. Established ideas from functional programming too, such as immutability and persistent data structures [**25**] have also been gaining greater prominence (e.g. immutability in the shared-memory community [**6**]), not least because of Clojure's influence. A final (older) paradigm for concurrency, dataflow programming, bears similarities to functional programming and is well-described in [**26**] and [**27**].

These paradigms are not always implemented at the language level. There are also a considerable number of libraries. For Java, the java.util.concurrent library was added in JDK5 to aid programmers in development. Some libraries such as OpenMP, TBB (Thread Building Blocks) and Java's JDK7 ForkJoin library[**28**] [**29**] are designed to make shared-memory concurrency more declarative. However these are geared towards more 'embarassingly-parallel problems' (e.g. scientific computing) which can be separated into independent tasks easily and have simple coordination requirements. Recent languages Fortress, X10 and Unified Parallel C also fall under this category. This project's focus is rather on the complex coordination requirements of more typical concurrent applications. Libraries for message-passing include the well-established MPI (Message-Passing Interface) (which has bindings in several languages), MPJava [**30**] and Kilim [**31**] (both Java libraries).

One particular (Python) library, Kaemalia [**32**], uses a concurrency paradigm bearing similarities to that of the language being developed. Further discussion is given in section 3.4.

### 2.1.2 Tool Support

A lot of tool support exists for concurrency. Many tools use static or dynamic analysis of a program to expose concurrency bugs such as data races and deadlock. Tools such as ConTest [**33**], rsTest [**34**], CheckMate [**35**] and Chord [**36**] all perform dynamic analysis of Java programs. Most work by instrumenting the bytecode at synchronisation points. Intel Thread Checker and Sun Thread Analyzer [**37**] are examples of similar non-Java tools. Other tools include model checkers such as SPIN [**38**], CHESS [**39**] and Java PathFinder [**40**]. These may be used to prove correctness of (smaller) concurrent programs. [**8**] documents many more tools.

### 2.1.3 Theoretical Approaches

Theoretical approaches were not researched in detail. [**41**],[**42**] and [**43**] survey the area in depth. CSP (Communicating Sequential Processes) was the only approach examined. A selection of Java software library implementations of theoretical approaches include JCSP (CSP for Java), CTJ (Communicating Threads for Java) and Join Java.

## 2.2 Translator Implementation

The considered approaches to translator implementation are discussed. This is followed by closer examination of the chosen approach.

### 2.2.1 Considered Approaches

One approach allows one to specify new constructs to add to the Java language: the Java Language Extender (JLE) [**44**]. An advantage is that development time to construct a translator is likely to be reduced. However it emphasises adapting Java to specific problem domains. It was felt that it might not be flexible enough if at any point more fundamental changes to the language were required.

A more flexible approach considered was to modify the source code of the OpenJDK compiler itself [**45**][**46**]. Unfortunately its size and complexity, and the time required to gain familiarity with it also ruled out this choice.

### 2.2.2 Chosen Approach

Figure 34 illustrates the high-level approach chosen. This involved the use of the ANTLR parser generator [**47**] [**48**]. The benefits of this approach were previous experience with ANTLR and the availability of an open source ANTLR grammar for the Java language [**1**]. The Java Compiler API is used to programmatically invoke javac on the Java code generated by the translator.

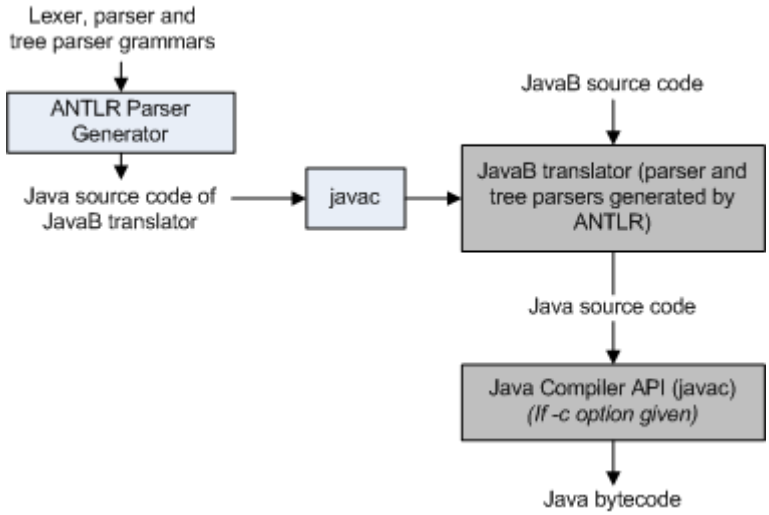


**Figure 1 - Chosen approach to translator implementation. Shows the use of ANTLR to generate the translator written in Java and javac to compile it. The rest of the diagram shows how to use the translator to translate a JavaB file into a Java file (and optionally bytecode).**

The OpenJDK compiler source code was studied on occasion during development of the ANTLR grammars.

#### 2.2.2.1 Summary of ANTLR

ANTLR generates recursive-descent recognisers (lexers, parsers, tree parsers) from ANTLR grammar specifications. These above recognise different input types: character streams, token streams, abstract syntax trees (ASTs), respectively, and form the different phases of the translator. Tree parsers may generate ASTs for further processing or generate textual output.

A commonly used feature of ANTLR are *actions*. These are custom pieces of code written in Java often used to perform various semantic checks. Different phases often have different actions, reflecting the purpose of that phase. In this translator actions also aid the code generation phase.

During translator development, three main ANTLR resources were consulted: two books concerning ANTLR and language implementation [**48**] [**49**] and also the antlr-interest mailing list[2].

Interestingly, an automated approach to tree construction [**50**] was discovered after implementation. Unfortunately, it had two major drawbacks. Firstly, not all types of AST rewrite rules were supported. Secondly, although it can allow different actions to be performed in each phase, it is not amenable to changes to the AST itself between phases.

[2] http://www.antlr.org/mailman/listinfo/antlr-interest

# 3.

# Language Definition

This chapter explains the precise semantics (and syntax) of the language developed. Implementations of classic 'toy' concurrency problems such as Producer-Consumer are used to illustrate the language ideas. The language semantics presented here form the requirements specification for the translator implementation. Examples are used throughout to aid understanding of the language concepts[3]. The chapter closes with a brief comparison with other similar existing languages/approaches.

It should be noted that the language is still in its infancy. In particular, the Switch component's semantics are not clarified and there are also many other standard components yet to be implemented which are discussed in Appendix C.

[3] All the examples in this chapter except those in sections Copy Synchronisation Primitive and Switch Synchronisation Primitive may be successfully run through the translator (provided on the attached DVD-ROM) and the generated output compiled (javac) and run (java). Appendix B provides additional examples that may be run through the translator. Appendix H provides a system manual for using the translator.

## 3.1 Core Concepts

### 3.1.1 IntProducer-IntConsumer Example

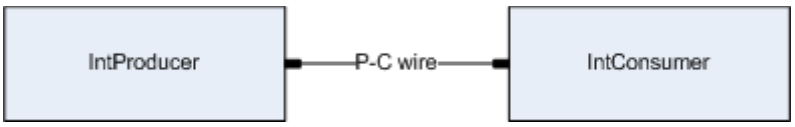


**Figure 2 - components, wires and boundaries in producer-consumer example**

```
import java.util.Random;

component IntProducer { // producer that produces integers
    boundary right int out!; // (A) boundary declaration(s)
    private Random rand = new Random(); // (B) internal state of component

    __run__ { // (C) run method of component
        while(true) {
            int produced = produce_item();
            out![produced]; // (D) synchronisation statement
        }
    }

    out![int val] { // (E) the out! boundary's corresponding handler
        __block__;
    }

    private Integer produce_item() { // (F) ordinary Java method
        return rand.nextInt(1000);
    }
}
```

**Code Listing 1 - Component Definition for the IntProducer component**

```
component IntConsumer { // consumer that consumes integers
    boundary left int in?; // (A) boundary declaration(s)

    __run__ { // (B) run method of component
        while(true) {
            int consumed = in?; // (C) synchronisation statement
            consume_item(consumed);
        }
    }

    in?[int val] { // (D) the in? boundary's corresponding handler
        __block__;
    }

    private void consume_item(int value) { // (E) ordinary Java method
        System.out.println("IntConsumer received the value "+value);
    }
}
```

**Code Listing 2 - Component Definition for the IntConsumer component**

```
public class Application {
    public static void main(String[] args) {
        composition c1 = IntProducer.IntConsumer; // Composition
declaration of a sequential composition of IntProducer and IntConsumer
        __start__ c1;
    }
}
```

**Code Listing 3 - Wiring code that wires (instances of) the two components together and then 'starts' them (see section 3.1.4 [Wiring components])**

## 3.1.2 Basic Terminology

### 3.1.2.1 Components

Components are class-like structures, given by a component definition (e.g. Code Listing 1 and Code Listing 2), whose instances can be likened to "threaded objects" since they both possess internal state and run within their own thread of control (they have a run method) (except passive components; see section 3.2.2). Components declare explicit *boundaries* which may be thought of as the interface by which the component may be 'wired to' other components' boundaries.

During program execution, components may directly 'synchronise' with each other on such wired boundaries. This involves the sending and receiving of values (primitives or objects) between the two boundaries of the components (which could be ignored if the value itself is unimportant). Such a synchronisation is initiated by a component using a *synchronisation statement* (e.g. `in?` or `out![produced];`).

### 3.1.2.2 Boundaries

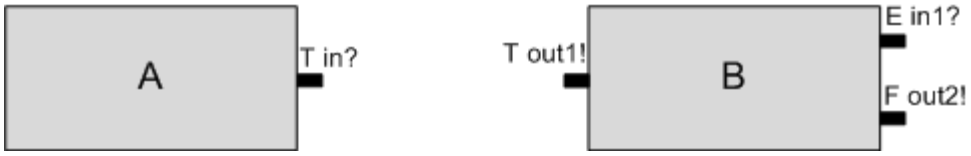


**Figure 3 - Components and boundaries. Component A has single *right* boundary (in?) which is an 'inward' boundary of type T. Component B has three boundaries. Its *left* boundary out1! is an 'outward' boundary also of type T. A's in? and B's out1! boundaries are compatible and may be wired together.**

Boundaries have a type (primitive or Object), specifying the type of the values to be sent or received, and a direction (*in* (?) or *out* (!)), specifying whether the component is sending or receiving on that boundary. Boundaries also have a 'side', *left* or *right*, on which they are relative to their component (seen in Figure 3). Boundaries of components being wired together must be compatible. They must have the same type (e.g. int with int) and opposite directions (*in?* with *out!*) and sides (*left* with *right*). In the IntProduer-IntConsumer example, IntProducer's right *out!* boundary is compatible with IntConsumer's left *in?* boundary, allowing them to synchronise if wired together (and 'started'; see section 3.1.4.2). Note that side and direction of boundaries is independent. Inward and outward boundaries may appear on either side of a component.

A component may have multiple boundaries, allowing it to synchronise with multiple components (potentially simultaneously). In IntProducer-IntBufferCell-IntConsumer example of section 3.2.1, IntBufferCell has two boundaries, one which is wired with IntConsumer and another which is wired with IntProducer.

### 3.1.2.3 Wires

Wires act as the basic means of connecting components. Components themselves are independent entities. The way they are connected defines the system behaviour. As well as a basic means of connection, wires provide the necessary synchronisation semantics. These semantics are described in section 3.1.3.

#### 3.1.2.4 Tug

*Tug* refers to when a component executes a synchronisation statement (e.g. `out![produced]`) on a certain boundary and thus attempts to synchronise with another component. The terms pushing/pulling are used synonymously. They explicitly indicate the direction of the boundary.

The following fragment from Code Listing 1 shows this:

```
__run__ { // (C) run method of component
    while(true) {
        int produced = produce_item();
        out![produced]; // (D) synchronisation statement
    }
}
```

**Code Listing 4 - IntProducer's run method. At (D) IntProducer is said to be tugging on its out boundary. The term 'tugging' tends to assume the boundary of interest in the component is wired to a boundary of another component.**

Thus for IntProducer above, tugging means sending/pushing the produced item (on its `out` boundary) on the wire.

#### 3.1.2.5 Handlers

Every boundary of a component has a corresponding handler. A handler specifies what action (i.e. statements) the other component wired to that boundary should take, if they are the *first* component to tug on the wire (seen more clearly in section 3.1.3).

Essentially, the first tugging component runs the other component's corresponding boundary handler. Examples of handlers will be seen shortly.

### 3.1.3 Synchronisation on a Wire

The semantics of how a synchronisation takes place on a wire is presently described by way of two examples, which illustrate the two different cases that can take place; whether the executed handler blocks or runs to completion.

#### 3.1.3.1 Control flow of a Synchronisation in IntProducer-IntConsumer Example

Figure 4 illustrates the course of events for one type of synchronisation that can occur; namely where a handler action specifies to block. The corresponding IntProducer and IntConsumer JavaB code is relisted:

```
import java.util.Random;

component IntProducer { // producer that produces integers
    boundary right int out!; // (A) boundary declaration(s)
    private Random rand = new Random(); // (B) internal state of component

    __run__ { // (C) run method of component
        while(true) {
            int produced = produce_item();
            out![produced]; // (D) synchronisation statement
        }
    }

    out![int val] { // (E) the out! boundary's corresponding handler
        __block__;
    }

    private Integer produce_item() { // (F) ordinary Java method
        return rand.nextInt(1000);
    }
}
```

**Code Listing 5 - Component Definition for the IntProducer component (relisting of Code Listing 1)**

```
component IntConsumer { // consumer that consumes integers
    boundary left int in?; // (A) boundary declaration(s)

    __run__ { // (B) run method of component
        while(true) {
            int consumed = in?; // (C) synchronisation statement
            consume_item(consumed);
        }
    }

    in?[int val] { // (D) the in? boundary's corresponding handler
        __block__;
    }

    private void consume_item(int value) { // (E) ordinary Java method
        System.out.println("IntConsumer received the value "+value);
    }
}
```

**Code Listing 6 - Component Definition for the IntConsumer component (relisting of Code Listing 2)**

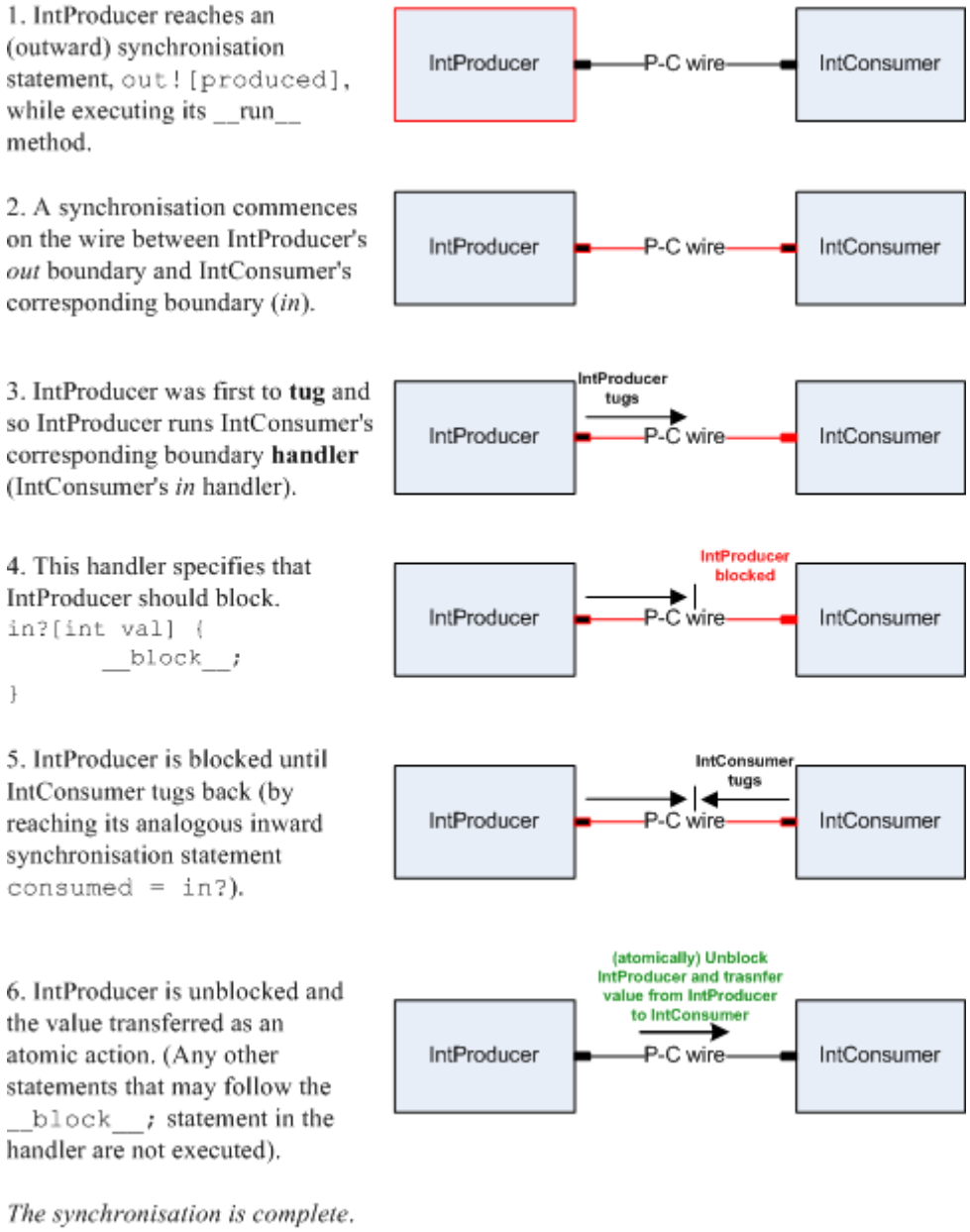


**Figure 4 - A synchronisation scenario for the IntProducer-IntConsumer example, where IntProducer is first to tug. Parts of the diagram are highlighted to indicate what is happening at each point in the synchronisation.**

Note that if step 1 changed so that IntConsumer reached its `in?` statement first (and thus was first to tug), then this would essentially reverse the diagram so that IntConsumer runs the corresponding boundary in IntProducer (which is also defined to block).

Two components cannot initiate a synchronisation at exactly the same time; one component always tugs first and thus runs the other component's handler. For many cases, handlers are defined to just block (as above).

### 3.1.3.2 Control flow of a synchronisation in IntProducer-IntEater Example

Figure 5 shows the course of events for the second type of synchronisation that can occur; where a handler action does not block but is run to completion without blocking. IntEater is equivalent to IntConsumer except that its `in?` handler is empty.

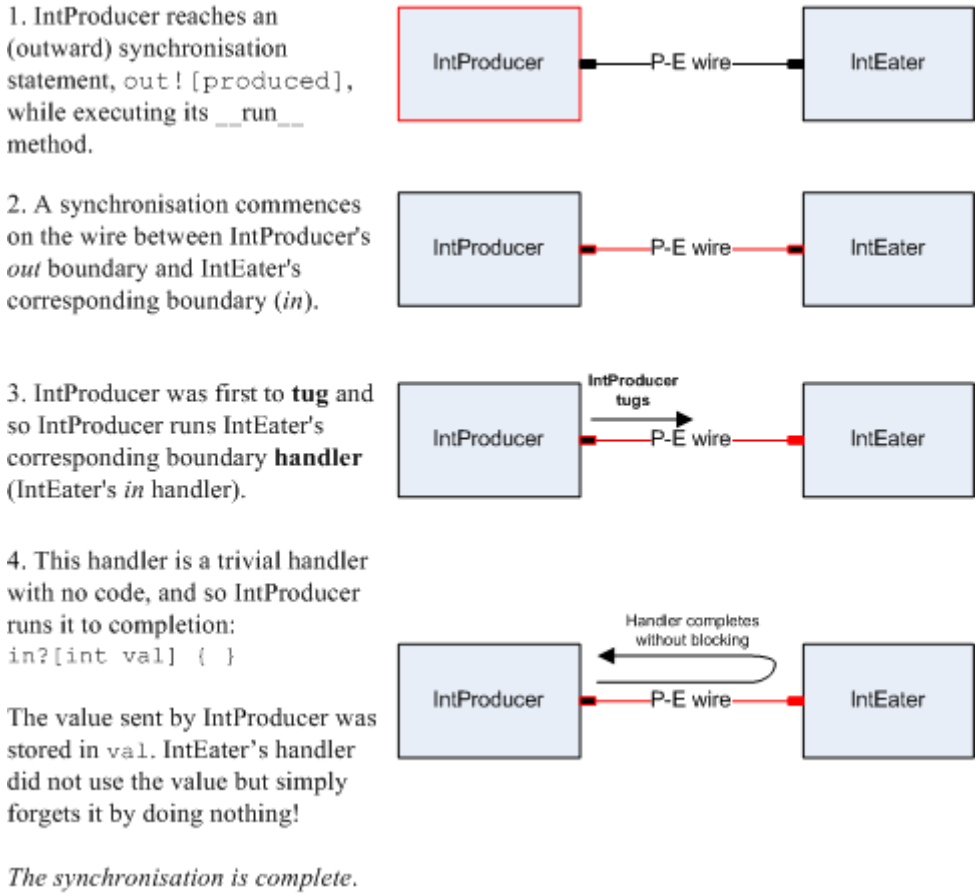


**Figure 5 - A synchronisation scenario for the IntProducer-IntEater example, where IntProducer is first to tug. (In fact, this is the only possible scenario because IntEater is a passive component which never tugs - see section 3.2.2 for explanation of passive components) Parts of the diagram are highlighted to indicate what is happening at each point in the synchronisation.**

#### 3.1.3.3 Completing a synchronisation

The above two examples represent the two alternative ways of completing a synchronisation after a component initiates a synchronisation (tugs first). Either:

1. The component runs the handler and is blocked. When the other component tugs back, it *atomically* unblocks and sends/receives the value (Figure 4) (two-component participation), or
2. The component runs the handler, and executes it to completion without blocking (Figure 5) (single-component participation). The value is sent/received by means of the *handler parameter* (see section 3.2.4).

Also, in the possible scenario that a currently non-tugging component (A) starts to tug *whilst* another already tugging component (B) is running A's handler, then A should wait to determine whether the B blocked or completed A's handler.

### 3.1.4 Wiring Components (Wiring/Glue Code)

An essential requirement for the language is the ability to wire the boundaries of components together. The significance of boundary *types*, *directions* and *sides* comes to a forefront; only compatible boundaries may be wired together (see section 3.1.2.2).

### 3.1.4.1 Sequential Composition ( . )

This section uses the following wiring code:

```
public class Application {
    public static void main(String[] args) {
        composition c1 = IntProducer.IntConsumer; // Composition
declaration of a sequential composition of IntProducer and IntConsumer
        __start__ c1;
    }
}
```

**Code Listing 7 - Wiring code for IntProducer-IntConsumer example (relisting of Code Listing 3)**

The Sequential Composition operator is used to perform wiring.

Sequential composition wires up the *right* boundaries of its left operand's with the corresponding *left* boundaries of its right operand (the 'inner' boundaries). Thus the order that boundaries are defined within a component matters (specifically, the order with respect to other boundaries of the *same* side matters). The number of corresponding boundaries must be the same and each 'boundary pair' must also be compatible with each other (otherwise it is a type error). Figure 6 illustrates this for Code Listing 7 above:

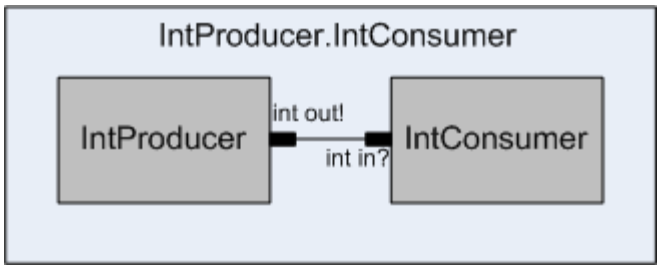


**Figure 6 - Sequential composition of IntProducer and IntConsumer components**

The result of a sequential composition is (an example of) a *composition component* / 'supercomponent'. Ordinary components and composition components may be treated uniformly. A composition component is defined by the components composing it rather than by an explicit component definition.

The left boundaries of the composition component are formed from the left boundaries of the left operand and the right boundaries are formed from the right boundaries of the right operand (the 'outer' boundaries). This is shown in Figure 7:

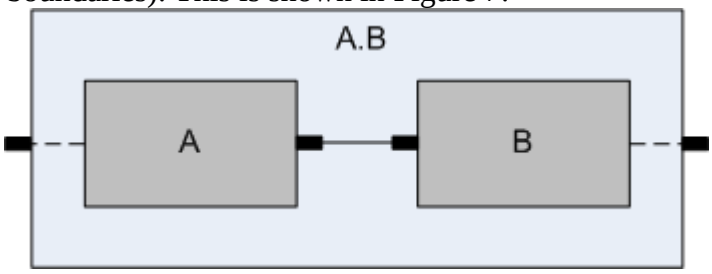


**Figure 7 - Sequential composition operator applied to its two operand components A and B. The dashed lines illustrate how the 'outer' boundaries of the operands form the boundaries of the resulting component. (Boundary names, types and directions are not shown).**

Composition components can be treated as black boxes. For example, Figure 7 could be depicted as:


**Figure 8 - A composition components may be treated as a black box, with only its 'boundary interface' visible to the outside world.**

Since a composition components are themselves components, they too can be operands in a sequential composition, as Figure 9 shows:

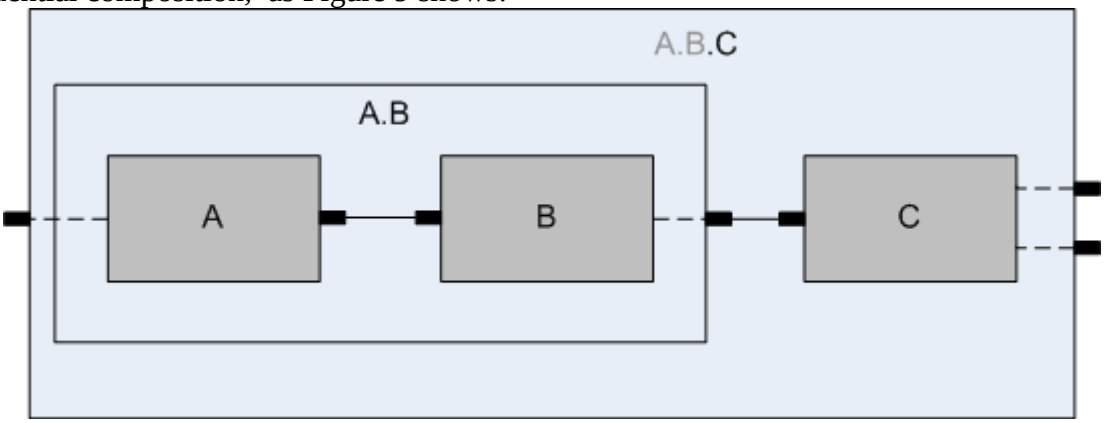


**Figure 9 - The result of applying the Sequential Composition operator to two components as its operands itself is a component (a composition component). Each box in the diagram represents a component. (Boundary names, types and directions are not shown).**

Composition components are discussed further in section 3.3.

### 3.1.4.2 'Starting' a Composition

A __start__ statement in the wiring code takes the referenced composition component and begins execution of the __run__ method of all *active* ordinary components 'within' that composition.

One typing constraint on starting a composition is that it has no 'outer'/'dangling' boundaries. Figure 6 is such an example. Figure 7 however, has boundaries remaining. A programmer can artificially 'close' these remaining boundaries if necessary by using the trivial components discussed in Appendix C.

### 3.1.4.3 Achieving Flexible Wiring

To wire up components whose boundaries differ only by their order, components must be redefined with different boundary orders to be compatible. This lack of flexibility is better resolved by use of a Twist component:

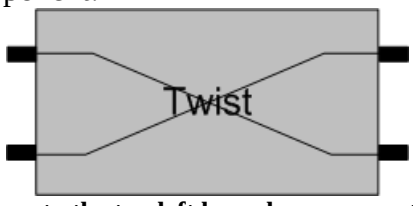


**Figure 10 - Twist component. Tugs to the top left boundary cause a tug on the wire connected to the bottom right boundary (and vice versa). Tugs to the bottom left boundary cause a tug on the wire connected to the top right boundary (and vice versa).**

Appendix C discusses this approach and an alternative approach in further detail. The Loopback component is also discussed as a means to achieve further flexibility.

## 3.2 A More Complex Synchronisation

This section outlines some more concepts by example.

### 3.2.1 IntProducer-IntBufferCell-IntConsumer Example

This example extends the previous IntProducer-IntConsumer to add an IntBufferCell component. The IntBufferCell has internal state to hold an integer and to mark its state as either empty/not-empty.


**Figure 11 - components, wires and boundaries in producer-IBC-consumer example**

```
component IntBufferCell {
    // boundaries
    boundary left int in?;
    boundary right int out!;

    // internal state
    boolean empty = true;
    int value = 0;

    in?[int val] { // here 'val' is an *input parameter*
        if(empty) {
            value = val;
            empty = false;
        }
        else {
            out![value];
            value = val;
        }
    }

    out![int val] { // here 'val' is a *return parameter*
        if(!empty) {
            val = value;
            empty = true;
        }
        else {
            val = in?;
        }
    }
}
```

**Code Listing 8 - Component Definition for the IntBufferCell component. It differs from previous examples in that it is a passive component. Additionally, its handlers contain no __block__ statement. Tugging components may however block via chains of synchronisations.**

```
//P.IBC.C
public class Application {
    public static void main(String[] args) {
// wire IntProducer's right 'out' boundary to IBC's left 'in' boundary,
// and wire IBC's right 'out' boundary to IntConsumer's 'in' boundary
        composition c = IntProducer.IntBufferCell.IntConsumer;
        __start__ c;
    }
}
```

**Code Listing 9 - Wiring code that wires (instances of) the three components together and then 'starts' them. The component definitions for IntProducer and IntConsumer are given in Code Listing 1 and Code Listing 2, respectively. IntBufferCell is given in Code Listing 8.**

The wiring code for this application is similar to the IntProducer-IntConsumer example, with an extra sequential composition operator required.

An example of a synchronisation for this example is shown in Figure 12 and Figure 13:

**1.** IntConsumer reaches a synchronisation statement, `in?`, whilst executing its __run__ method.


**2.** A synchronisation commences on IBC-C wire between IntConsumer's *in* boundary and IntBufferCell's corresponding boundary (*out*).


**3.** IntConsumer was first to tug and so IntConsumer runs IntBufferCell's corresponding boundary handler (IntBufferCell's *out* handler).


**4.** This handler's behaviour depends on the current value of IntBufferCell's empty variable:

```
out[int ! val] {
    if(!empty) {
        val = value;
        empty = true;
    }
    else {
        val = in?;
    }
}
```

If the IBC holds a value, then the value sent to IntConsumer is the held value.

Otherwise, the IBC attempts to tug on its other boundary, *in*, to retrieve a value from its neighbouring IntProducer to send.

Assume the second case.

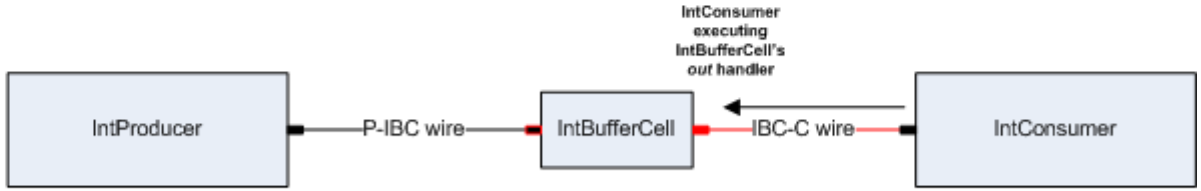


**Figure 12 - A possible synchronisation for the IntProducer-IntBufferCell-IntConsumer example (top half).**

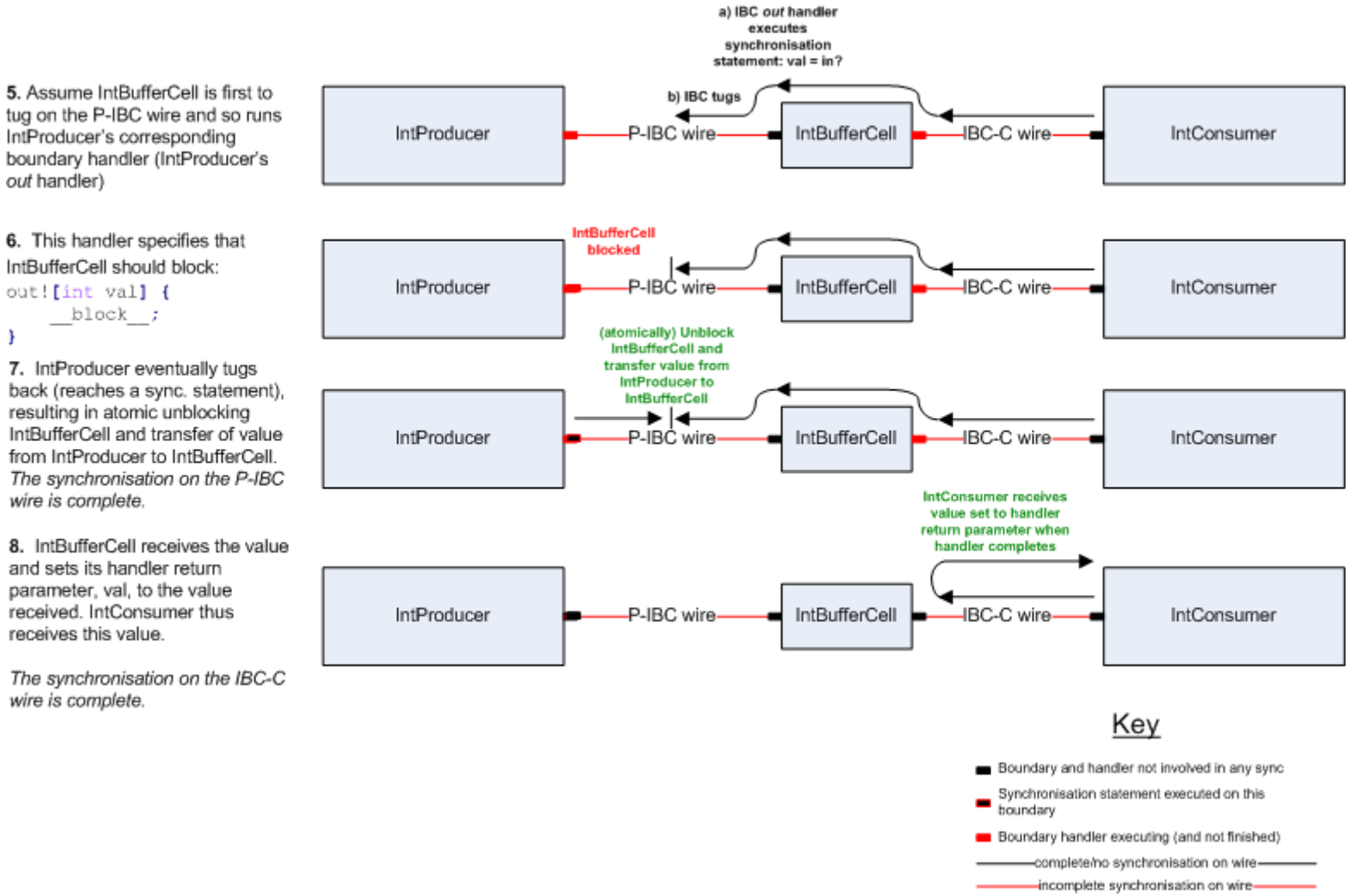


**Figure 13 - Bottom half of a synchronisation for the IntProducer-IntBufferCell-IntConsumer example.**

### 3.2.2 Active and Passive Components

IntBufferCell is an example of a *passive* component. Components may be either *active* or *passive*. Active components are those with a __run__ method which is executed by their own thread of control. Passive components do not. They only contain *handlers* (and internal state). These handlers are (sometimes) run by other components trying to synchronise with the component (see section 3.1.3.1).

### 3.2.3 Chains of Synchronisations

The example synchronisation above introduced the possibility of chains of synchronisations. This takes place because handlers may also execute synchronisation statements; not just the run methods of components. Both IntBufferCell's handlers contained synchronisation statements.

### 3.2.4 Handler Parameters

This example also demonstrated handler parameters (e.g. `val`). These may be input parameters or return parameters, used in inward or outward boundary handlers, respectively. Thus in inward handlers, the handler parameter should only ever be read. Likewise, in an outward handler, the handler parameter should only ever be written to (e.g. step 4 in Figure 12). A handler may also ignore the handler parameter completely (Appendix B.1 offers an example of this (IntBufferEater)). Also note that handlers may only have a single parameter, corresponding to the single value being transferred in a synchronisation.

## 3.3 Composability

Section 3.1.4.1 discussed one operator to create composition components: Sequential Composition. Another operator that adds far more power to the way components may be composed together is Tensor Composition.

### 3.3.1 Tensor Composition ( #[4] )

Tensor composition does *not* wire components as sequential composition does. Tensor places its left operand component above its right operand component. The resulting composition component's left boundaries is the left boundaries of its left operand followed by the left boundaries of its right operand (analogously for its right boundaries). Figure 14 shows this:

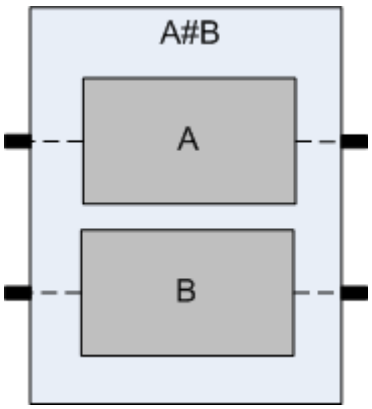


**Figure 14 - Tensor composition of components A and B. As the left operand, A's boundaries come before B's boundaries in the resulting (composition) component. The tensor composition operator performs no wiring; it simply creates a new component that is the 'vertical sum' of its parts.**

[4] An alternative symbol that may be used for the tensor operator is '*/' (without quotes).

Again, as with sequential composition components, the resulting component may be treated uniformly like any other component and so be further tensored or sequentially composed with other components. An example follows:

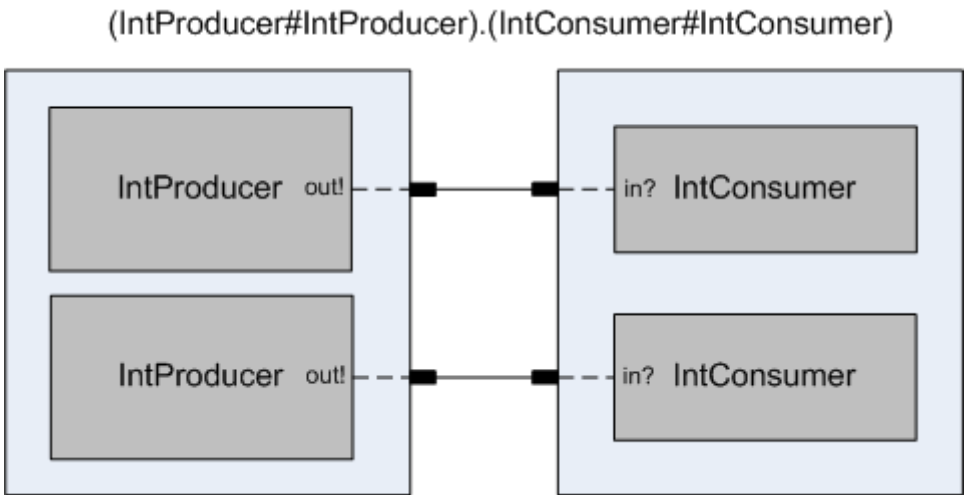


**Figure 15 - Example of a sequential composition of two tensor composition components. Each corresponding pair of boundaries are joined by a wire.**

Here, the components resulting from two tensor compositions are sequentially composed.

A further example shows a single DoubleIntConsumer that receives values on two boundaries from separate IntProducers:

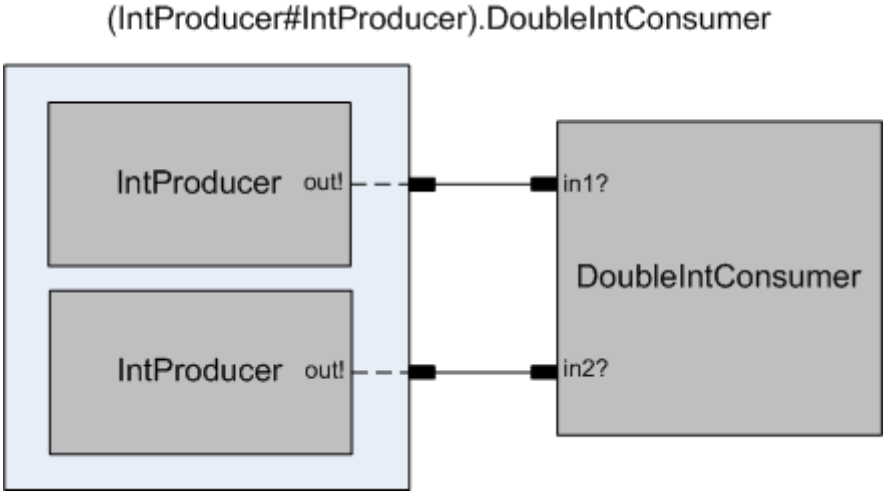


**Figure 16 - Further example of sequential composition of two tensored componenents (IntProducers) with a single ordinary component (DoubleIntConsumer) that can receive on two boundaries.**

## 3.4 Copy Synchronisation Primitive

The basic synchronisation provided by an ordinary wire is sufficient for some applications. However, more sophisticated synchronisation semantics are often required. This section and the next introduce two synchronisation primitives which allow a single synchronisation to take place among three parties rather than just two.

Copy allows a broadcast/copy of a value to take place from one boundary to two boundaries, as Figure 17 shows:

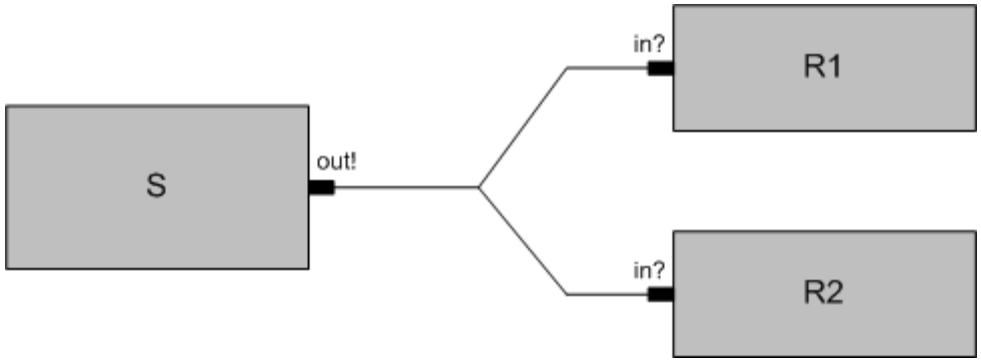


**Figure 17 - Sender S's out! boundary wired via a 'splitter' wire to the in? boundaries of receivers R1 and R2. To be precise, the two receiving boundaries do not necessarily have to belong to two separate components. It could be a single component with two receiving boundaries.**

From an implementation standpoint, a desirable way to achieve this would be to introduce a special Copy *component* that encapsulates the required synchronisation semantics of copy (which follow shortly) and use sequential composition to wire it with its left and right neighbouring components, as illustrated in Figure 18:

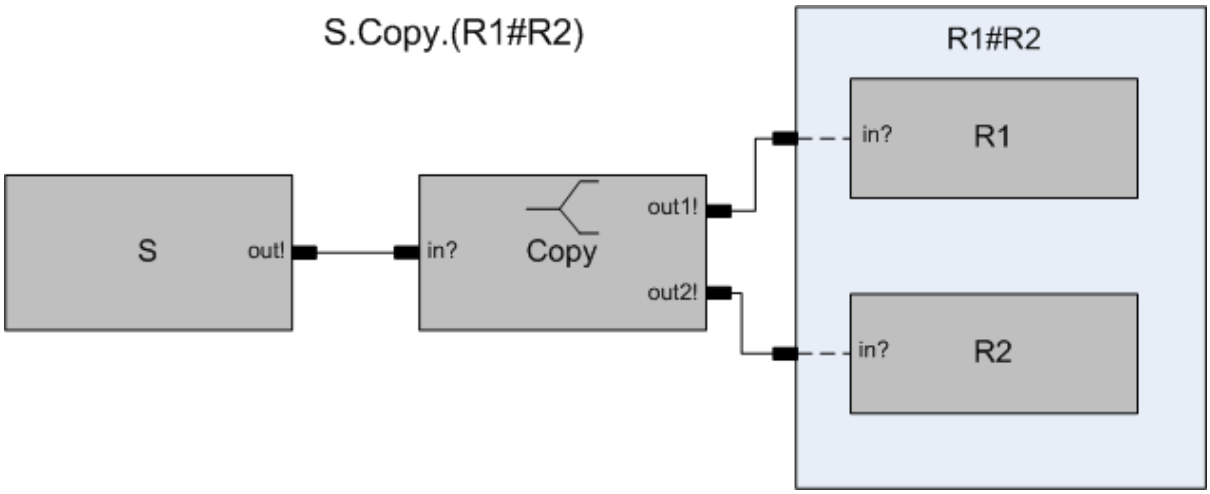


**Figure 18 - Special Copy component that provides the desired synchronisation semantics. Strictly, S and Copy should be surrounded with a box (the left-associativity of . dictates that S be sequentially composed with Copy *first* and then the resulting composition sequentially composed with R1#R2).**

The current implementation of Copy does not take this approach, mainly for reasons of time constraints (see section 4.1.3.5).

### 3.4.1 'Direction' of Copy

Copy actually has two cases, depending on the boundary directions; either there are *two senders* or *two receivers*. Viewing Copy again independently from its possible implementation (i.e. using a Copy component), Figure 19 shows this:

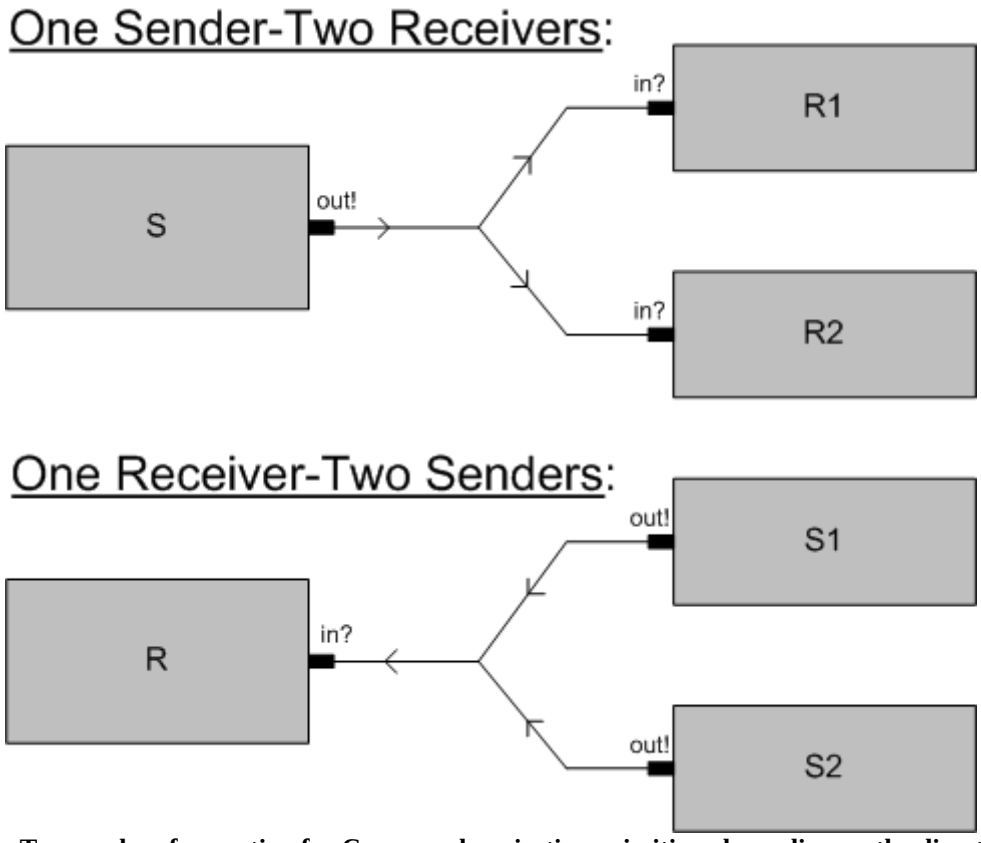


**Figure 19 - Two modes of operation for Copy synchronisation primitive, depending on the direction of the boundaries connected to it. The semantics of Copy differ for each.**

The intuitive case is where there are two receivers: the value is copied to both. The case of two senders is rather different. The synchronisation only completes when both senders are sending the same value. This latter case has not been explored in detail.

## 3.4.2 One Sender-Two Receivers

### 3.4.2.1 Semantics

The semantics of the one sender-two receiver case set up depends on whether a sender or a receiver is first to tug on the wire:

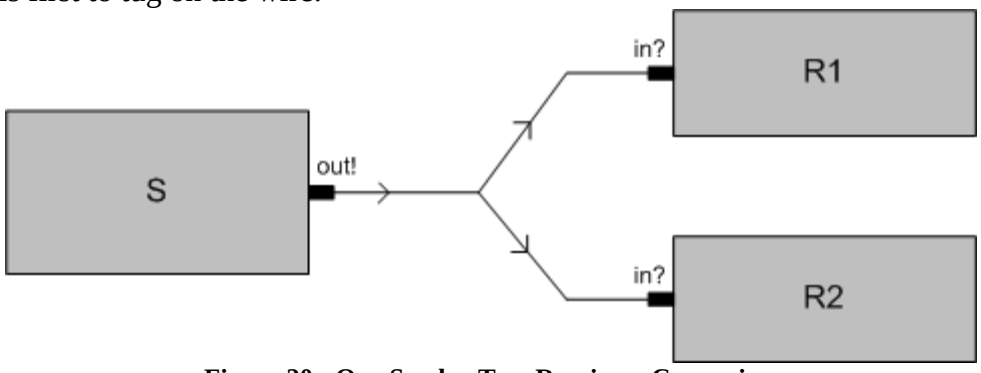


**Figure 20 - One Sender-Two Receivers Copy wire.**

For clarity of explanation, R1 and R2 are assumed to be two separate components. Strictly, however, there could be a single component with two inward (receiving) boundaries.

*In general:*
When the first component tugs (not limited to S), the handlers of *both* the other components are run (possibly concurrently). These separate interactions may be treated as individual 'sub-synchronisations'. Even if one sub-synchronisation completes (by the handler blocking and then being unblocked, or by completing without blocking; see section 3.1.3.3), the components involved must wait until the other sub-synchronisation also completes. S' send value is only (atomically) transferred when *both* sub-synchronisations have completed (completion as described in section 3.1.3.3).

*S tugs first:*
When S tugs first, R1 and R2's handlers are run (beginning a sub-synchronisation with R1 and R2). Once *both* sub-synchronisations have completed, the entire synchronisation is complete, and the value may be transferred. Value transfer actually takes place in one of two ways. Either the value is transferred upon unblocking of a blocked handler, or the value is transferred via the handler parameter. (If via the latter, a desirable property is that the value itself is not be made visible until the entire synchronisation completes. This is one area of Copy's semantics that are unclear).

*Rx tugs first:*
When Rx (R1 or R2) tugs first, the semantics are similar to when S tugs first. The only difference is that Rx *must* run S's handler *before* Ry's handler. The value being sent must be known before Ry's handler is run.

### 3.4.2.2 Examples

In Figure 21 below, when IntProducer tugs, it runs *both* the other two parties' handlers. In this example, both handlers block. The value is transferred to both atomically *only when* both IntConsumers have tugged back. When an IntConsumer tugs first, the same actions take place except that the sender's (IntProducer) handler *must* be run before the other receiver's (IntConsumer) handler.

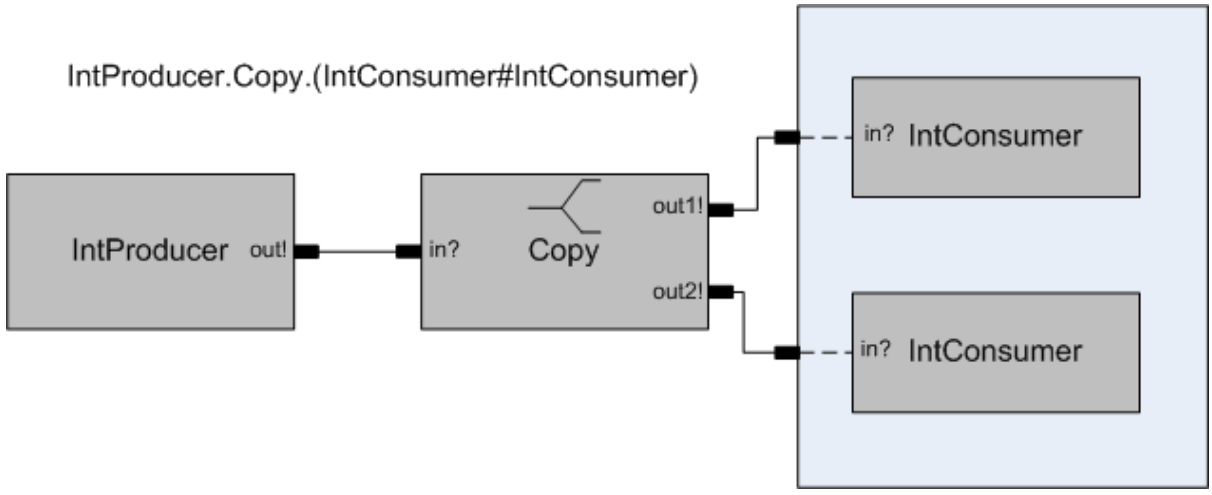


**Figure 21 - One IntProducer sender and two IntConsumers as receivers with a Copy component acting as conceptual Copy wire between them**

The following example illustrates the semantics of Copy when a handler completes without blocking. It also shows how the above example may be extended to include asynchronous communication by using an IntBufferCell:

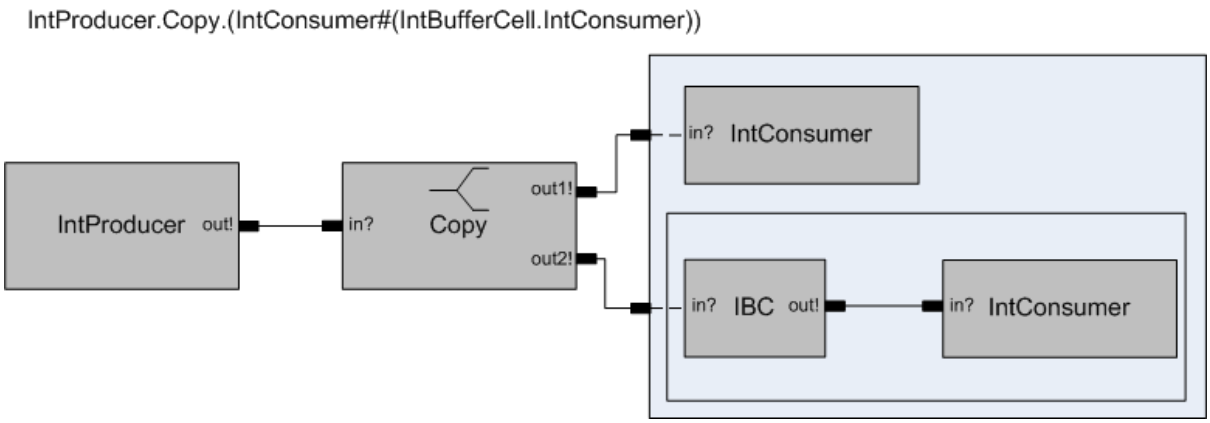


**Figure 22 - Achieving asynchrony in the bottom IntConsumer by making the IntBufferCell the second receiver of the Copy.**

If IntProducer tugs first, both the top IntConsumer's and IntBufferCell's handlers are run. Even though IntBufferCell's handler completes (and thus the IntProducer-IntBufferCell sub-synchronisation also completes), the entire synchronisation only completes once the top IntConsumer tugs back, unblocking its blocked handler. The lower IntConsumer does not directly participate in the synchronisation; the IntBufferCell does so 'on its behalf'.

### 3.4.3 One Receiver-Two Senders

This direction bears similarity to above. Again, the entire synchronisation may only complete when both sub-synchronisations complete. However, in addition to that, the synchronisation only completes when both senders are sending the same value. Further details of this direction of operation have not been explored.

## 3.5 Switch Synchronisation Primitive

Switch enforces mutual exclusively access to one boundary by two competing boundaries, as Figure 23 shows:

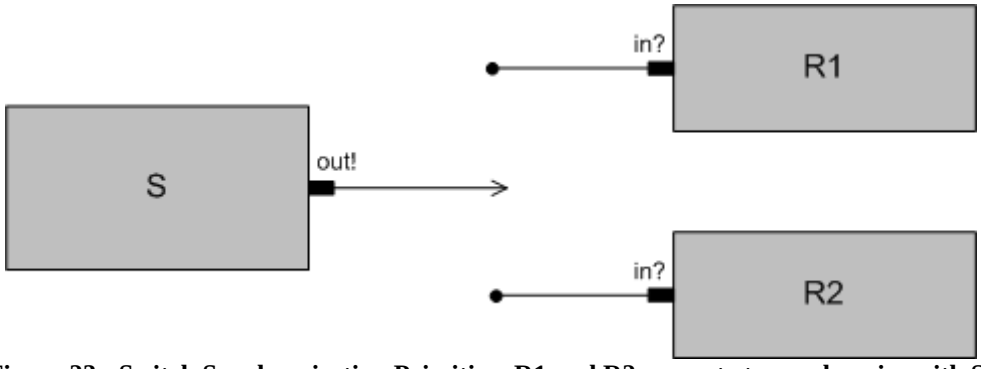


**Figure 23 - Switch Synchronisation Primitive. R1 and R2 compete to synchronise with S.**

Similar to Copy, a special Switch *component* that encapsulates the required synchronisation semantics would be used. This is illustrated in Figure 24:

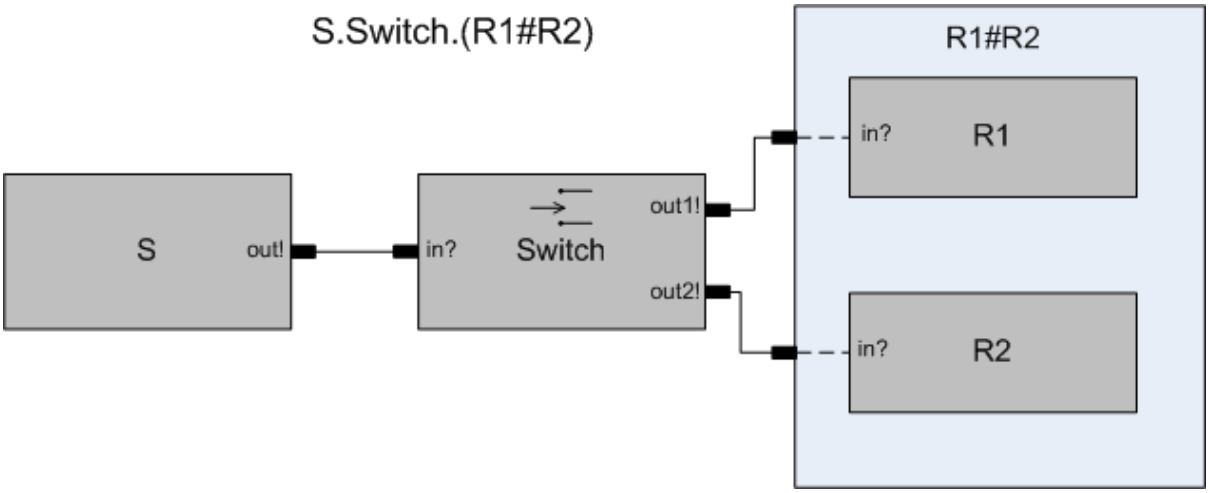


**Figure 24 - Special Switch component that provides the desired synchronisation semantics.**

Switch is an example of an unimplemented construct. Its semantics have not yet been established and require more investigation.

## 3.6 Language Comparisons

This final section gives a brief comparison with existing languages.

### 3.6.1 Comparison with Kamaelia

Kamaelia has a very similar concept of components [**51**]:

> *Components have "inboxes" and "outboxes" through with they communicate with other components.*
>
> *A component may send a message to one of its outboxes. If a linkage has been created from that outbox to another component's inbox; then that message will arrive in the inbox of the other component. In this way, components can send and receive data - allowing you to create systems by linking many components together.*
>
> *Each component is a microprocess - rather like a thread of execution.*

One unique feature of the language that differs from Kamaelia is the use of handlers. With handlers, a chain of synchronisations can take place among components (as in producer-IBC-consumer).

### 3.6.2 Comparison with CSP (Communicating Sequential Processes)

In contrast to CSP, communication between components (processes) is tightly controlled. When a wire connects two components, only those two components can use it. It cannot be used by other components to communicate. CSP channels on the other hand can be read or written to by any process that has a ‘handle’ on that channel. Thus even though the superficial resemblance of wires to CSP channels makes communication appear like message-passing, the communication style is actually a disciplined form of shared-memory, where the wire connecting two components is the shared-memory between those components. (Sending a value on a wire corresponds to an atomic write. Receiving a value on a wire corresponds to an atomic read).

# 4.

# Translator Design and Implementation

The design and implementation details of the translator are discussed in this chapter. Prior to this the core translation mechanisms between JavaB and Java are examined. In particular, the classes implementing the synchronisation semantics of ordinary and copy wires are examined.

## 4.1 Translation Mechanisms

### 4.1.1 Translation Classes

A 'manual' translation of what the translator might generate for the IntProducer-IntConsumer and IntProducer-IBC-IntConsumer examples was initially undertaken. This was a necessary requirement to understanding what code the translator should generate. It also ironed out some misconceptions in the language semantics. The manual translation was written in a way such that it could be generated by the translator, so that the translation mechanism could apply to any program written in the language.

The translation closely follows the conceptual ideas of the language, with Java classes such as Boundary, Wire and HandlerRunnable representing boundaries, wires and handlers, respectively. IntProducer, IntConsumer and IntBufferCell likewise represent their respective components. Component is a superclass of all components. Active components implement Runnable (passive components do not). The Wire classes (NormalWire and CopyWire) implement the required semantics of a synchronisation on a wire. These contain most of the complex logic and (Java) synchronisation (see section 4.1.3).

Table 1 lists and describes the various classes. Figure 25 shows the corresponding class diagram.

| Class/Interface (I) | Purpose | Explanation |
|---|---|---|
| **'Fixed' classes (written once; not generated by translator)** | | |
| Component | Represents a component. | The translator generates component classes such as IntProducer that subclass this class.<br><br>It is an abstract class which stores the name of the component and an explicit lock associated with the component. Subclasses can access this lock via a call to |

| | | |
|---|---|---|
| | | getLock(). |
| Boundary<T> | Represents a boundary of a component. | The application code creates these by invoking create_boundary_x(), a method generated for each boundary of a Component. This is used instead of natively instantiating Boundary objects in the wiring code so that the Boundary's associated handler can be created *inside the Component. Handlers require access to the internal state of the Component* and thus the Boundary object, whose constructor requires the handler code, is also instantiated inside the Component.<br><br>The generic parameter T is the type of the boundary (in the JavaB sense). |
| HandlerRunnable<T> (Interface) | Represents a handler.<br><br>The HandlerRunnable's run() method contains the handler's (partially translated) code. | HandlerRunnable is a modification to java.util.concurrent.Runnable that allows an input parameter and a return parameter, both of generic type T (the type of the handlers Boundary). The send() method in the Wire class implementations will pass the value being sent as an input parameter and ignore the return parameter. Similarly, the receive() method will not pass any meaningful input parameter but will use the return parameter.<br><br>When a call to the create_boundary_x(Wire<T>) method of a Component owning boundary x is made, an anonymous HandlerRunnable object (handler) is instantiated and then passed as a parameter into the Boundary constructor to create the Boundary object representing x. |
| Wire<T> (Interface) | Represents any type of wire; specifies the public interface all wires must have.<br><br>Implementation of translated run method and handlers of a component becomes simpler, because the translated component does not need to know what underlying Wire implementation is being used to wire its boundaries. | send() and receive() methods correspond to JavaB's synchronisation statements (e.g. `in?` and `out![value]`). send() is called precisely when there is an outward boundary synchronisation statement (e.g. my`Out![value]`). receive() is called precisely when there is an inward boundary synchronisation statement (e.g. `myIn?`). send() and receive() can be called by either component run methods or handlers.<br><br>The blockHandler() and finishHandler() are (only) called by handlers (HandlerRunnables). They correspond to the two ways a handler may complete (see section 3.1.3.3). Essentially they inform the Wire (and thus the second tugger) of the outcome of handler execution (*blocked* vs. *finished without blocking*) so that the second tugger knows what to do (*unblock the other tugger* vs. *start a new synchronisation*). blockHandler() also implements the required blocking behaviour.<br><br>All Wires must implement a setBoundaries() method. The purpose of this method is given in the |

| | | |
|---|---|---|
| | | ApplicationProdCons explanation. |
| NormalWire<T> | Implements required synchronisation semantics of an 'ordinary' wire. | Explanation same as for Wire<T>.<br><br>In addition, the setBoundaries() method is of the form:<br>setBoundaries(sender,receiver)<br>since there is one sending boundary and one receiving boundary. |
| CopyWire<T> | Implements required synchronisation semantics of a copy wire. | Explanation same as for Wire<T>.<br><br>In addition, the setBoundaries() method is of the form:<br>setBoundaries(sender,receiver1,receiver2)<br>since there is one sending boundary and **two** receiving boundary. |
| **(Examples of) Translator-generated classes** | | |
| IntProducer | Subclass of Component. Translation from IntProducer component definition. | This component has one boundary, *out!*, which is translated into the private Boundary object *out* (which is instantiated when the wiring code in ApplicationProdCons calls create_boundary_out(Wire<Integer> wireAttachedTo)). In general, a create_boundary_x() method is generated for each boundary x of a component.<br><br>The out![produced] synchronisation statement in the run method is translated into a call to send() on the Wire wireAttachedTo object.<br><br>The __block__; statement in the out! handler is similarly translated to a call to blockHandler() on the Wire wireAttachedTo object. |
| IntConsumer | Subclass of Component. Translation from IntConsumer component definition. | This component's translation is very similar to IntProducer. It too has a single boundary, in?, which is translated in the same way to a private Boundary object *in* (ApplicationProdCons calls create_boundary_in(Wire<Integer> wireAttachedTo)).<br><br>The in? synchronisation statement in the run method is translated into a call to receive() on the Wire wireAttachedTo object.<br><br>The __block__; statement is translated in exactly the same way as in IntProducer. |
| ApplicationProdCons | The main application containing the translated wiring code. | This code starts by instantiating all required (translated) component instances, then instantiates the required Wire objects (e.g. NormalWire), then (indirectly through calls to create_boundary_x(Wire<T>)) instantiates the required Boundary objects.<br><br>The Wire objects need to have a reference to the Boundary objects representing each end of the wire. |

| | | |
|---|---|---|
| | | Thus the wiring code also has calls to setBoundaries(sender,receiver) on each of the Wire objects. The constructor of Wire is not used for passing these parameters because Boundary and Wire require a mutual reference to each other, and thus one must be instantiated before the other (Wire before Boundary). |

**Table 1 - The name, purpose and explanation of all classes used in a manual translation. The 'fixed' classes are those classes that are standard and are not ever generated by the translator. These classes, together with the generated component and wiring code / application classes, implement the required semantics of the JavaB language**

The next section gives an example translation that uses these classes.

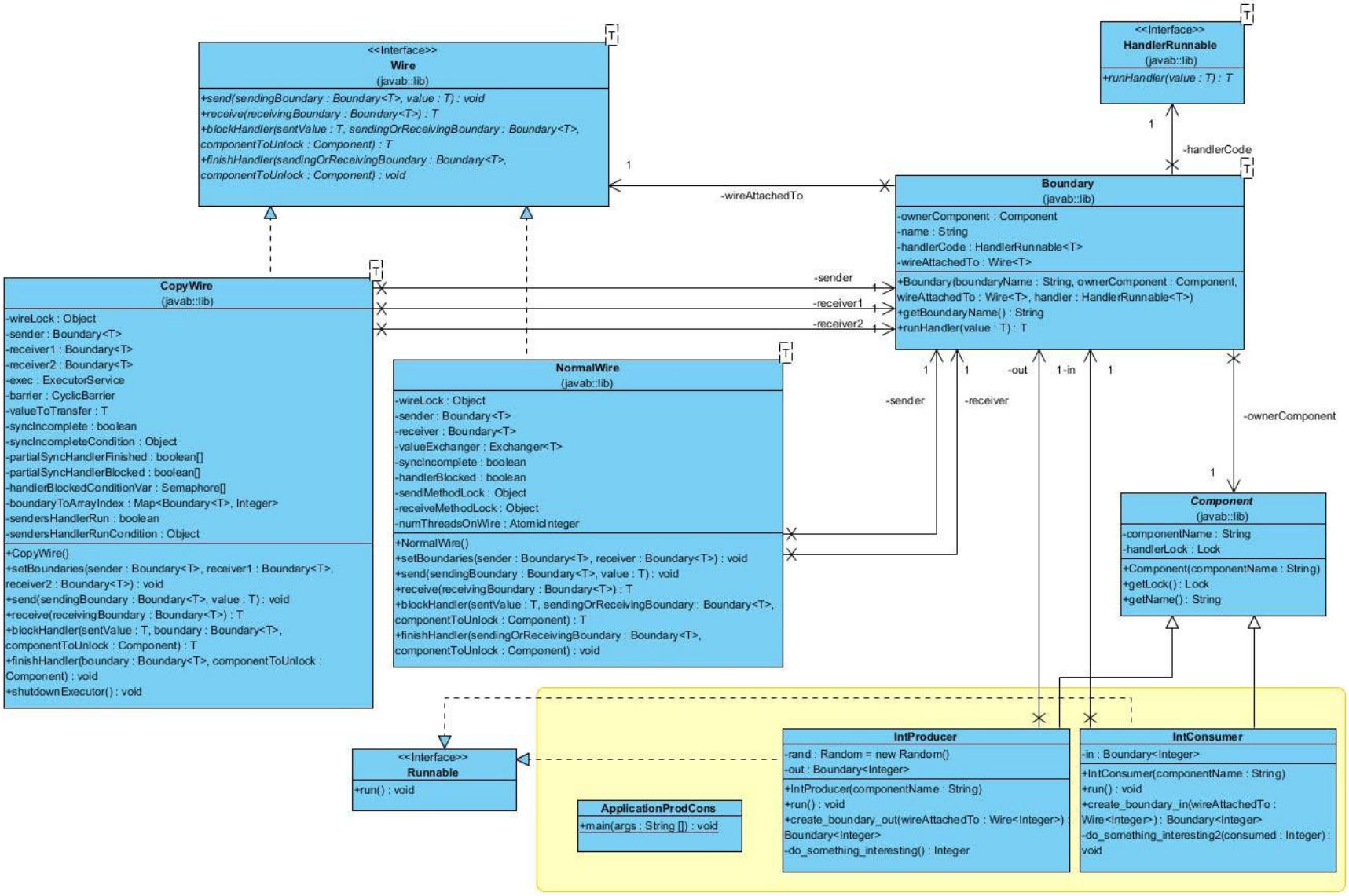


**Figure 25 - Classes used in translation. The classes in yellow are the classes that are generated by the translator. The other classes are fixed. In this example, the IntProducer and IntConsumer classes are translated from their component definitions from Chapter 3.**

### 4.1.2 Translation of IntProducer-IntConsumer Example

The translations of IntProducer.javabc, IntConsumer.javabc and ApplicationProdCons.javab from section 3.1.1, are subsequently described. A further example is given in Appendix E.

#### 4.1.2.1 IntProducer.java and IntConsumer.java

The translations are given below, followed by an explanation.

```
import javab.runtime.*;

import java.util.Random;

public class IntProducer extends Component implements Runnable {
    public IntProducer() {
        super("IntProducer"); // pass name of component to superclass
(Component)
    }

    // INTERNAL STATE
    private Random rand = new Random();

    // BOUNDARIES
    private Boundary<Integer> out;

    // HANDLERS
    public Boundary<Integer> create_boundary_out(Wire<Integer>
wireAttachedTo) {
        // the handler for this boundary

        HandlerRunnable<Integer> handler = new HandlerRunnable<Integer>() {
            public Integer runHandler(Integer val) {
                // no translator housekeeping code required before user
code

                // "user code" (with JavaB parts translated) -- which could
contain a (translated) 'block;' statement
                // a block; statement in a handler is replaced with the
following single line that blocks and when unblocked returns immediately
with the value received
                if(true) return
out.getWireAttachedTo().blockHandler(val,out,IntProducer.this); // 'block;'

                // translator housekeeping code following the user code (if
user code blocks then this code is unreachable)

out.getWireAttachedTo().finishHandler(out,IntProducer.this); // At this
point we know that we have finished the handler without blocking (i.e. the
sync is complete, apart from the housekeeping tasks we are about to do now)
                return val;
                // IF OUTWARD HANDLER: it doesn't matter that we're
returning back the value the sender gave us as our dummy value for the
exchanger; the sender will ignore it anyway
                // IF INWARD  HANDLER: the handler (return) parameter val
should have been set by the programmer; if it never gets set by the
programmer then the (dummy) value that was passed in will be returned
            }
        };
```

```
        // create boundary (name, owner component, wire, handler)
        out = new Boundary<Integer>("out", this, wireAttachedTo, handler);
        return out;
    }

    // RUN METHOD
    public void run() { // (C) run method of component
            while(true) {
                int produced = produce_item();
                out.getWireAttachedTo().send(out,produced); //
out![produced] // (D) synchronisation statement
            }
        }

    // OTHER METHODS
    private Integer produce_item() { // (F) ordinary Java method
            return rand.nextInt(1000);
        }
}
```

**Code Listing 10 - IntProducer.java - the translation of the component definition IntProducer.javabc**

```
// consumer that consumes integers
import javab.runtime.*;

public class IntConsumer extends Component implements Runnable {
    public IntConsumer() {
        super("IntConsumer"); // pass name of component to superclass
(Component)
    }


    // BOUNDARIES
    private Boundary<Integer> in;

    // HANDLERS
    public Boundary<Integer> create_boundary_in(Wire<Integer>
wireAttachedTo) {
        // the handler for this boundary

        HandlerRunnable<Integer> handler = new HandlerRunnable<Integer>() {
            public Integer runHandler(Integer val) {
                // no translator housekeeping code required before user
code

                // "user code" (with JavaB parts translated) -- which could
contain a (translated) 'block;' statement
                // a block; statement in a handler is replaced with the
following single line that blocks and when unblocked returns immediately
with the value received
                if(true) return
in.getWireAttachedTo().blockHandler(val,in,IntConsumer.this); // 'block;'

                // translator housekeeping code following the user code (if
user code blocks then this code is unreachable)
                in.getWireAttachedTo().finishHandler(in,IntConsumer.this);
// At this point we know that we have finished the handler without blocking
(i.e. the sync is complete, apart from the housekeeping tasks we are about
to do now)
                return val;
```

```
                // IF OUTWARD HANDLER: it doesn't matter that we're
returning back the value the sender gave us as our dummy value for the
exchanger; the sender will ignore it anyway
                // IF INWARD  HANDLER: the handler (return) parameter val
should have been set by the programmer; if it never gets set by the
programmer then the (dummy) value that was passed in will be returned
            }
        };


        // create boundary (name, owner component, wire, handler)
        in = new Boundary<Integer>("in", this, wireAttachedTo, handler);
        return in;
    }

    // RUN METHOD
    public void run() {
        while(true) {
                    int consumed = in.getWireAttachedTo().receive(in);
                    consume_item(consumed);
                }
    }

    // OTHER METHODS
    public void consume_item(int value) {
            System.out.println("IntConsumer received the value "+value);
        }
}
```

**Code Listing 11 - IntConsumer.java - the translation of the component definition IntConsumer.javabc**

Many of the translation mechanisms in component definitions are straightforward. For example, both `IntProducer` and `IntConsumer` are components, and so extend `Component`. Moreover, both are active and so implement `Runnable`. Boundary declarations are translated into `Boundary` instance variables, with their type as a generic parameter. In both examples, boundaries were of type `int`; the translation process autoboxes them into their equivalent reference type, `Integer`.

Synchronisation statements are translated to calls to `send()` and `receive()` (for outward and inward synchronisations, respectively) on the Wire attached to the boundary being synchronised on.

Handlers are less trivial. They are not simply translated into methods. The `Wire` implementations do not know anything of the components they are attached to. They only know the `Boundary`s at each of their end-points. Wire thus invokes handlers through the appropriate `Boundary`. Therefore the handler must be defined before being passed into the `Boundary` constructor. This is performed in the `create_boundary_x()` methods.

#### 4.1.2.2 ApplicationProdCons.java

The wiring code translation is now given:

```
import javab.runtime.*;


import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.CountDownLatch;
```

```
public class ApplicationProdCons {
    public static void main(String[] args) {
         // Composition declaration of a sequential composition of
IntProducer and IntConsumer
        // create component instances contained in the composition
        IntProducer intProducer1 = new IntProducer();
        IntConsumer intConsumer1 = new IntConsumer();


        // create NormalWire and CopyWire instances
        NormalWire<Integer> WIRE_intProducer1_out_TO_intConsumer1_in = new
NormalWire<Integer>();


        // create boundary objects
        // (Boundary objects don't refer to each other, they only refer to
the Wire they are on the end of. That Wire object also has a mutual
reference to the Boundary object.)
        Boundary<Integer> intProducer1_out =
intProducer1.create_boundary_out(WIRE_intProducer1_out_TO_intConsumer1_in);
        Boundary<Integer> intConsumer1_in =
intConsumer1.create_boundary_in(WIRE_intProducer1_out_TO_intConsumer1_in);


        // now that we have created boundaries, set boundaries of the wire
object(s)
WIRE_intProducer1_out_TO_intConsumer1_in.setBoundaries(intProducer1_out,
intConsumer1_in);


        /* Start threads of all active components (implement Runnable) */
        // use a latch 'start gate' to ensure they start at the same time -
- see JCIP chapter 5
        final CountDownLatch startGate = new CountDownLatch(1);

        // add all Runnables to a set to be iterated over
        Set<Runnable> runnables = new HashSet<Runnable>();
        runnables.add(intProducer1);
        runnables.add(intConsumer1);


        // set of latch-altered Runnables that have been turned into
Threads
        Set<Thread> threads = new HashSet<Thread>();

        // iterate over them and wrap their run methods to include
startGate.await() at the beginning
        for(final Runnable r : runnables) {
          Thread t = new Thread() {
            public void run() {
              try {
                startGate.await();
                r.run();
              }
              catch(InterruptedException e) { e.printStackTrace(); }
            }
          };
          threads.add(t);
          t.start(); // also start the thread (it will await at latch)
        }
```

```
        // GO! (release all the threads)
        startGate.countDown();
    }
}
```

**Code Listing 12 - ApplicationProdCons.java - translation of wiring code ApplicationProdCons.javab**

The translated wiring code first instantiates the required `Component` and `Wire` objects. In the JavaB code, instances of components were *implicitly* constructed (the programmer does not have to create component instances).

For the actual wiring, a `Wire` must know all the boundaries on its end-points. Equally, a `Boundary` must know the `Wire` it is connected to. Thus a mutual reference between `Wire` and `Boundary` is required. This is achieved by constructing the `Wire` object first, (indirectly) passing that `Wire` into the `Boundary` constructor via a call to `create_boundary_x()` for each end-point boundary, and finally invoking `setBoundaries()` on the `Wire` with the returned `Boundary` objects.

Finally, all `Runnable` components are started, each assigned a thread. `CountDownLatch` is used to ensure threads start simultaneously.

### 4.1.3 Algorithms for Synchronisation Primitives

The algorithms of an ordinary wire (NormalWire) and Copy (CopyWire) are presented in this section. Time constraints meant Switch could not be implemented.

It was found that neither Copy and Switch could be implemented successfully just using component definitions, but instead required their own Wire, similar to NormalWire. It is still envisioned that special components will encapsulate these wires to look like ordinary components (see sections 3.4/3.5).

All wires implement send() and receive(). These methods correspond to outward and inward synchronisation statements, respectively. The blockHandler() and finishHandler() methods are called *by handlers* invoked during a synchronisation to indicate whether they blocked or completed.

#### 4.1.3.1 NormalWire Algorithm

```
shared (instance) variables:
Exchanger - synchroniser used to atomically exchange values between two
            threads (the two synchronising components). The first thread
            waits for the second to arrive at exchanger.
wireLock - both a lock and condition variable on handlerFinished or
           handlerBlocked events occuring
handlerBlocked - boolean used to reflect status of handler execution; that
                 the handler blocked. Set by the blockHandler() method.
handlerFinished - boolean used to reflect status of handler execution; that
                  the handler finished without blocking. Set by the
                  finishHandler() method.

// send() and receive() very similar (duals of each other)
send(T value) {
    atomically determine if caller first or second to tug on this wire
    if first then
        lock receiving component
```

```
        run receiving component's boundary handler
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()
    else // second tugger
        while(!handlerFinished and !handlerBlocked)
            wait on wireLock object
        if(handlerBlocked) // event was that handler blocked
            meet at Exchanger, passing value
        else // handlerFinished -- event was that handler finished
            reattempt synchronisation by calling send() recursively
        endif
        reset handlerBlocked and handlerFinished to false for next sync
    endif
}

T receive() {
    T valueReceived; // value to return to receiver

    atomically determine if caller first or second to tug on this wire
    if first then
        lock sending component
        run sending component's boundary handler
        valueReceived = return parameter value of that handler
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()
    else // second tugger
        while(!handlerFinished and !handlerBlocked)
            wait on wireLock object
        if(handlerBlocked) // event was that handler blocked
            meet at Exchanger, passing null // receiver is receiving, not
sending anything
            valueReceived = value received at Exchanger
        else // handlerFinished -- event was that handler finished
            reattempt synchronisation by calling send() recursively
            valueReceived = value returned from send()
        endif
    endif
}

/*
 * When a handler is invoked (by the first tugger), two events can occur:
 * 1. the handler blocks, in which case it calls blockHandler() to notify
 *  any second tuggers that may be waiting on wire.
 * 2. the handler finishes without blocking, in which case it calls
 *  finishHandler() to notify any second tuggers that may be waiting on
 *  wire.
 * Thus in a single synchronisation, only one of the two below methods is
 * called.
 */

blockHandler() { // called by first tugger's handler if it blocked
    unlock component the boundary handler belongs to (BEFORE BLOCKING via
the Exchanger)
    handlerBlocked = true
    notifyAll on wireLock
}

finishHandler() { // called by first tugger's handler if it finished
without blocking
    unlock component the boundary handler belongs to
    handlerFinished = true
```

```
    notifyAll on wireLock
}
```

**Figure 26 - Pseudocode of NormalWire Algorithm.**

Some points of interest include the use of java.util.concurrent.Exchanger. This class integrates the required blocking (of handlers) and value passing behaviour between sender and receiver.

Additionally, before the first tugger runs a handler, it acquires the owning component's lock is to ensure atomicity of handler execution with respect to other handlers of that component. Without this, state inconsistencies could arise due to race conditions when multiple separate synchronisations take place on the component's different boundaries.

#### 4.1.3.2 NormalWire Implementation

The Java implementation is given in Appendix F. Here however, a snippet of the if(firstToTug) block is shown. The pseudocode hid the complex details of if(firstToTug) that arose in the implementation due to a deadlock situation in NormalWire.

```
      // first to tug
      if(runTheHandler) {
        // possibility of not being able to acquire component's lock
        boolean done = false;
        while(!done) {
          // if we succeed in grabbing the lock
          if(receiver.getOwnerComponent().getLock().tryLock()) {
            receiver.runHandler(value);

            done = true;
          }
          // if we fail to grab the lock
          else {
            if(numThreadsOnWire.get() == 1) {
              Thread.yield(); // wait efficiently
            }
            if(numThreadsOnWire.get() == 2) {
              // pretend we were running a handler and blocked
              synchronized(wireLock) {
                this.handlerBlocked = true;
                wireLock.notifyAll();
              }

              // proceed to the exchange
              try { valueExchanger.exchange(value); }
              catch (InterruptedException e) { e.printStackTrace(); }

              // decrement numThreadsOnWire now that exchange/sync is done
              numThreadsOnWire.decrementAndGet();

              done = true;
            }
          }
        }
      }
```

**Code Listing 13 - The if(firstToTug)/if(runTheHandler) block from the send() method of NormalWire. It's complexity is much increased by the requirement to avoid deadlock. If no deadlock were possible, then all that would be required would be to acquire the receiving component's lock and run its handler (two lines of code!).**

The deadlock was possible when there was a component being tugged on two boundaries/wires simultaneously. If both tugs were the first tugs on their respective wires, then one of the tuggers acquired the component lock in order to run the component's handler.

The deadlock could occur when that handler itself tugs on the wire with the component that was just beaten to the component lock.

Sobocinski suggested a solution to resolve the deadlock. The use of tryLock() to attempt to acquire the component lock meant that failure to acquire the lock does not result in threads blocking. Instead, the loop ensures unsuccessful attempts are retried. numThreadsOnWire is used to know what action to take upon failure.

The high-level flow charts overleaf illustrate both the deadlock-prone and deadlock-free versions of the algorithm.

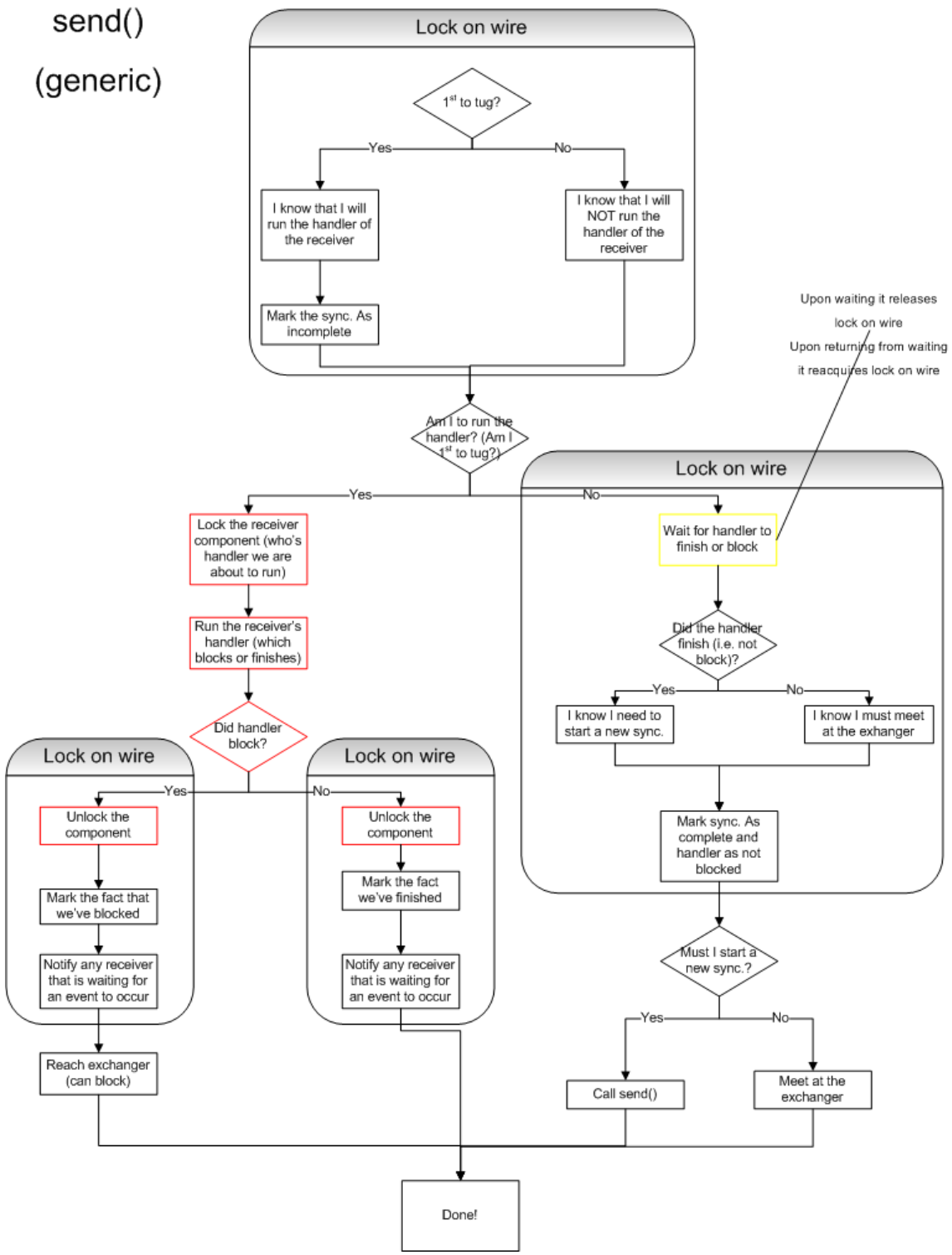


**Figure 27 - Flow chart showing logic of send() method of deadlock-prone NormalWire; the highlighted boxes show the source of deadlock. See section F.3.1 for an equivalent flow chart that includes code annotations from the actual implementation.**

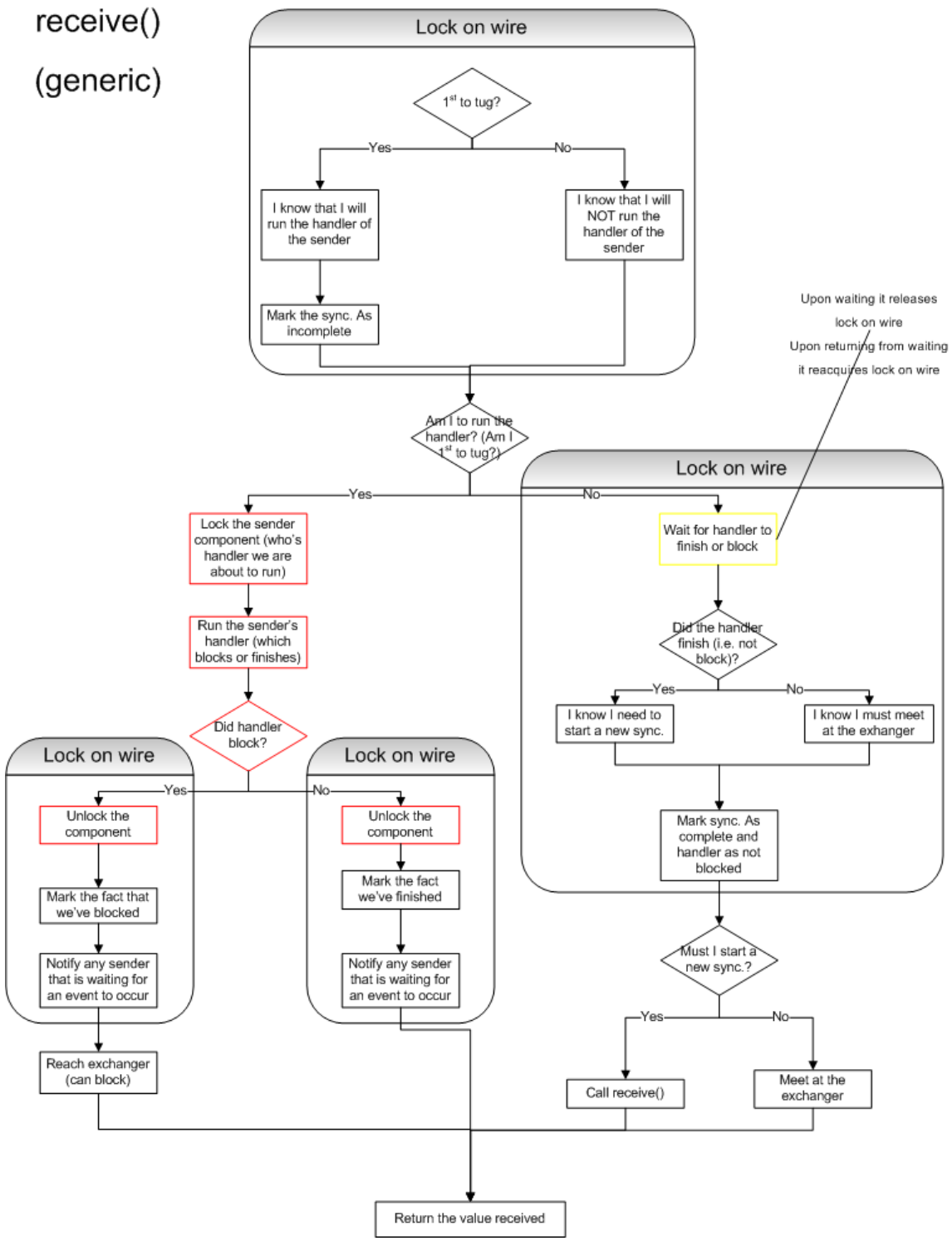


**Figure 28 - Flow chart showing logic of receive() method of deadlock-prone NormalWire; the highlighted boxes show the source of deadlock. See section F.3.1 for an equivalent flow chart that includes code annotations.**

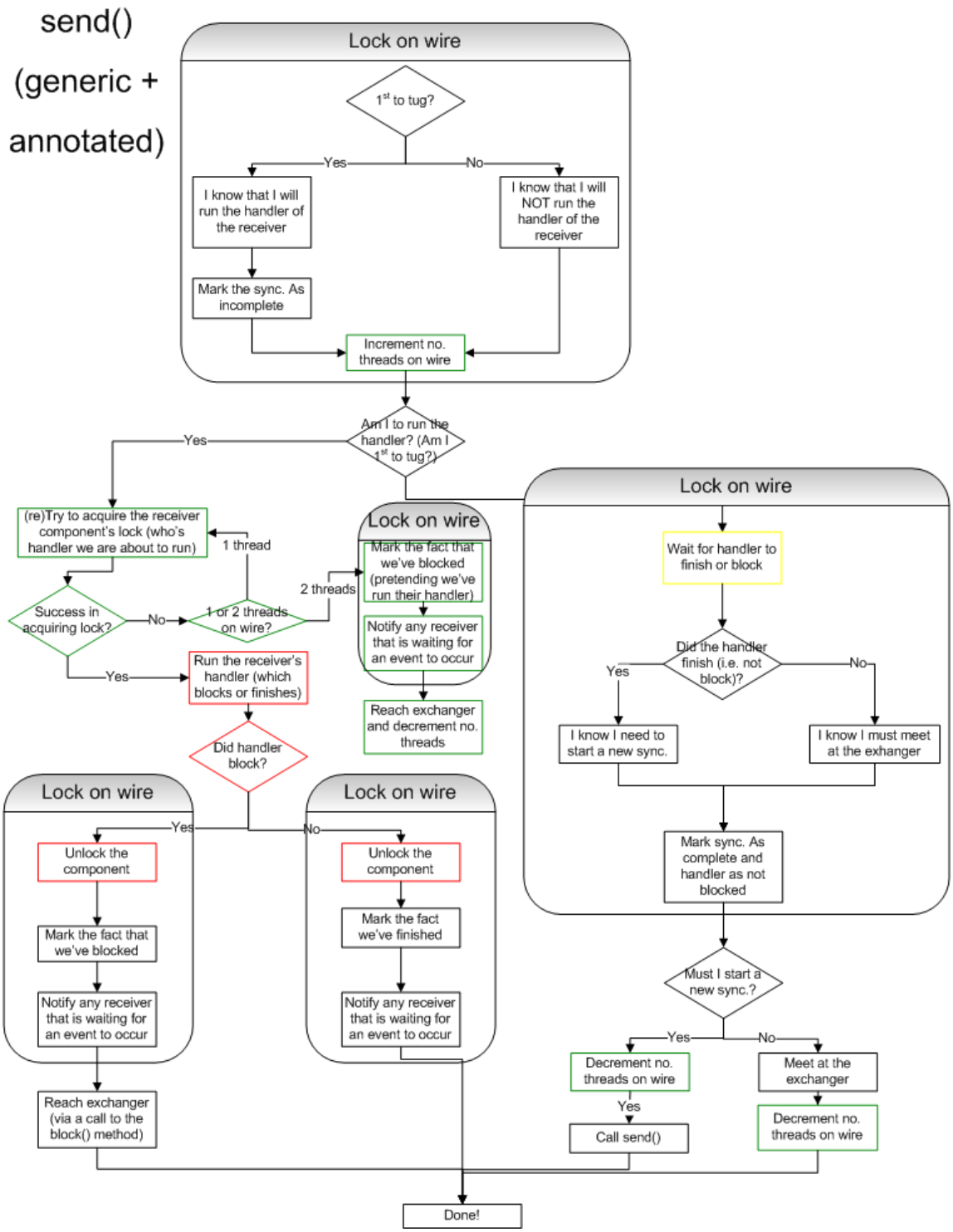


**Figure 29 - Flow chart showing logic of send() method of deadlock-free NormalWire; the added steps are highlighted in green. See section F.3.3 for an equivalent flow chart that includes code annotations.**

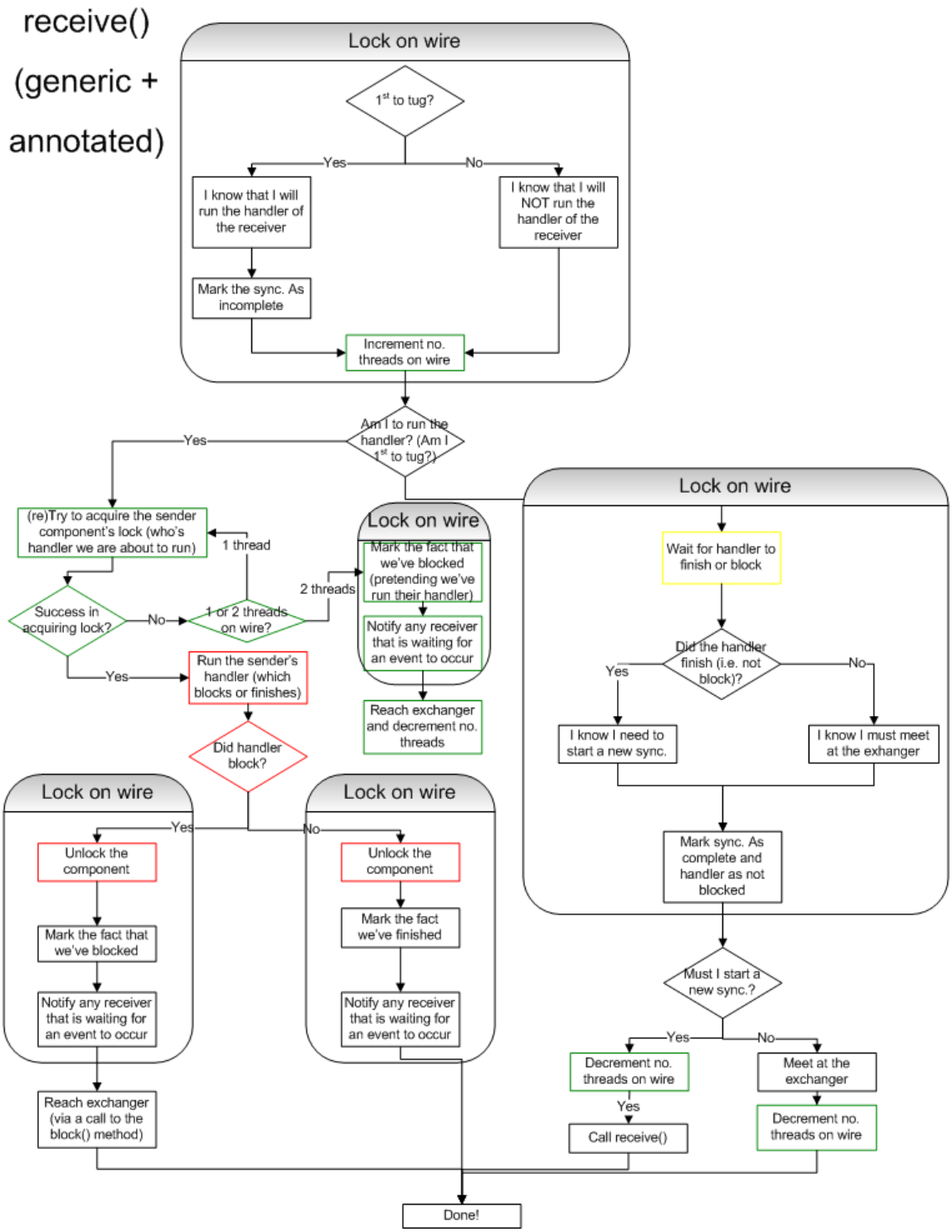


**Figure 30 - Flow chart showing logic of receive() method of deadlock-free NormalWire; the added steps are highlighted in green. See section F.3.3 for an equivalent flow chart that includes code annotations.**

#### 4.1.3.3 CopyWire Algorithm

An initial implementation of CopyWire was completed, which only considers the 'one sender-two receivers' case. The CopyWire pseudocode follows:

```
shared (instance) variables:
  wireLock - used as condition variable on changes to the state of
subSyncHandlerFinished or subSyncHandlerBlocked

  boolean[] subSyncHandlerFinished - array of booleans used to reflect
status of handler execution for each sub-synchronisation; that the handler
finished without blocking. Set by the finishHandler() method. The first
index is for the sender's boolean value; the second index is for
receiver1's boolean value; the third for receiver2's boolean value.

  boolean[] subSyncHandlerBlocked - array of booleans used to reflect
status of handler execution for each sub-synchronisation; that the handler
blocked. Set by the blockHandler() method. The first index is for the
sender's boolean value; the second index is for receiver1's boolean value;
the third for receiver2's boolean value.

  syncIncomplete - is the *entire* synchronisation complete

  valueToTransfer - value to be transferred

  barrier - Java synchroniser (CyclicBarrier), all threads must wait at
barrier before barrier released. The barrier also has an action that is
executed after all threads have arrived at the barrier but before the
barrier is released. This is used to ensure that the entire synchronisation
does not complete until both sub-synchronisations are complete.


initialisation {
  syncIncomplete = false // state of synchronisation is 'complete' (or,
ready to start a new sync.)
  set all boolean values of subSyncHandlerFinished to false // (no sub-
synchronisations are in mid-process)
  set all boolean values of subSyncHandlerBlocked to false // (no sub-
synchronisations are in mid-process)
}

barrier action {
  syncIncomplete = false // mark synchronisation as now being complete
  Notify any 'late' tuggers, who's handlers were run by the first tugger
and thus the 'late' tuggers participation is was not required; the late
tuggers just wait to try tugging again, the only try again when notified
}

// called by sender
send(T value) {
    valueToTransfer = value

  // set by this tugger in case they are a 'late' tugger, to allow them try
tugging again
  boolean startNewSync;
  do {
    startNewSync = false; // initailise/reset startNewSync

    atomically determine if caller first or second to tug on this wire
    if first { // first to tug
      spawn thread to run receiver1's handler {
```

```
        lock receiver1 component
        run receiver1 component's boundary handler, passing valueToTransfer
as handler input parameter
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()

        if handler finished without blocking, then wait at barrier in this
thread (if handler blocked, then the unblocking thread is responsible for
waiting at the barrier)
      }

      spawn thread to run receiver2's handler {
        lock receiver2 component
        run receiver2 component's boundary handler, passing valueToTransfer
as handler input parameter
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()

        if handler finished without blocking, then wait at barrier in this
thread (if handler blocked, then the unblocking thread is responsible for
waiting at the barrier)
      }

      wait at barrier
    }
    else { // second to tug
      first tugger (one of the receivers) must be running our handler
      therefore wait until this sender's handler has blocked or finished
      (CONDITION: subSyncHandlerFinished[sender] ||
subSyncHandlerFinished[sender])
      (CONDITION VARIABLE: wireLock)

      if handler blocked (i.e. first tugger blocked in my handler) {
        unblock first tugger
        wait at barrier (unblocking thread is responsible for waiting at
barrier)
      }
      elseif handler finished {
        startNewSync = true; // I'm 'late' to join synchronisation; set
flag for next time round the loop
        wait until current synchronisation completes (CONDITION:
!syncIncomplete)
      }

      subSyncHandlerFinished[sender] = false; // expected race condition in
resetting of flags
      subSyncHandlerBlocked[sender] = false;
    }
  } while(startNewSync);
}

/*
 * Called by either receiver.
 *
 * Logic is similar than that for send(); key difference is that sender's
handler is always executed before other
 * receiver's handler.
 *
 * Advantage to spawning separate threads for running sender's handler and
receiverY's handler is that it is general
```

```
 * enough that the barrier waiting mechanism works in all cases (normally
it would not make sense to have threads wait
 * 'sequentially' for each other -- purpose is so that each can reach the
barrier as separate threads).
 *
 * receiverX refers to the receiver that is currently running this method;
receiverY refers to the other receiver.
 *
 *   * (star) indicates new/different code to send() method.
 */
T receive() {
  // set by this tugger in case they are a 'late' tugger, to allow them try
tugging again
  boolean startNewSync;
  do {
    startNewSync = false; // initailise/reset startNewSync

    atomically determine if caller first or second to tug on this wire
    if first { // first to tug
      spawn thread to run sender's handler {
        lock sender component
        *run sender component's boundary handler*
        *set valueToTransfer to above executed handler's return parameter*
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()

        *notify other thread waiting for sender's handler / sub-
synchronisation to complete*

        if handler finished without blocking, then wait at barrier in this
thread (if handler blocked, then the unblocking thread is responsible for
waiting at the barrier)
      }

      // spawn separate thread for receiverY even though sender must
complete first
      spawn thread to run receiverY's handler {
        *wait until sender's handler / sub-synchronisation completes*

        lock receiverY component
        run receiverY component's boundary handler, passing valueToTransfer
as handler input parameter
        // unlocking of component not here, but occurs in blockHandler() or
finishHandler()

        if handler finished without blocking, then wait at barrier in this
thread (if handler blocked, then the unblocking thread is responsible for
waiting at the barrier)
      }

      wait at barrier
    }
    else { // second to tug
      first tugger (the sender or receiverY) must be running our handler
      wait until this receiver's handler has blocked or finished
      (CONDITION: subSyncHandlerFinished[thisReceiver] ||
subSyncHandlerFinished[thisReceiver])
      (CONDITION VARIABLE: wireLock)

      if handler blocked (i.e. first tugger blocked in my handler) {
        unblock first tugger
```

```
        wait at barrier (unblocking thread is responsible for waiting at
barrier)
      }
      elseif handler finished {
        startNewSync = true; // I'm 'late' to join synchronisation; set
flag for next time round the loop
        wait until current synchronisation completes (CONDITION:
!syncIncomplete)
      }

      subSyncHandlerFinished[sender] = false; // expected race condition in
resetting of flags
      subSyncHandlerBlocked[sender] = false;
    }
  } while(startNewSync);

  *return valueToTransfer;*
}

/*
 * When a handler is invoked (by the first tugger), two events can occur:
 * 1. the handler blocks, in which case it calls blockHandler() to notify
 *  any second tuggers that may be waiting on wire.
 * 2. the handler finishes without blocking, in which case it calls
 *  finishHandler() to notify any second tuggers that may be waiting on
 *  wire.
 * Thus in a single *sub-synchronisation*, only one of the two below
methods is
 * called.
 *
 * boundaryThatIsBlocking and boundaryThatIsFinishing could be the sender,
receiver1 or receiver2
 */

blockHandler(boundaryThatIsBlocking) { // called by handlers when they
blocked
    unlock component the boundary handler belongs to (BEFORE doing the
actual BLOCKING)
    subSyncHandlerBlocked[boundaryThatIsBlocking] = true
    notifyAll on wireLock
}

finishHandler(boundaryThatIsFinishing) { // called at the end of handlers
to indicate finishing without blocking
    unlock component the boundary handler belongs to
    subSyncHandlerFinished[boundaryThatIsFinishing] = true
    notifyAll on wireLock
}
```

**Code Listing 14 - Pseudocode of CopyWire Algorithm.**

Java's CyclicBarrier[5] synchroniser is used to ensure that whole synchronisation only completes when both sub-synchronisations complete. The tripping of the barrier corresponds to the synchronisation completing. A barrier action is used to notify 'late' tuggers that the synchronisation has completed so they may reattempt to tug.

[5] A barrier causes all threads that reach it to wait *until* all other threads have reached it also.

A do..while loop rather than recursion is used to implement the reattempt to tug. NormalWire could be improved to do the same. The use of a loop here is a more efficient and understandable mechanism than recursion.

Currently, there is a remaining problem of deadlock that occurs when running CopyWire.

#### 4.1.3.4 CopyWire Implementation

The implementation can be found in Appendix F.

#### 4.1.3.5 Implementing a Copy Component

As alluded to in section 3.4, use of a standard Copy component is currently unsupported in the translator:

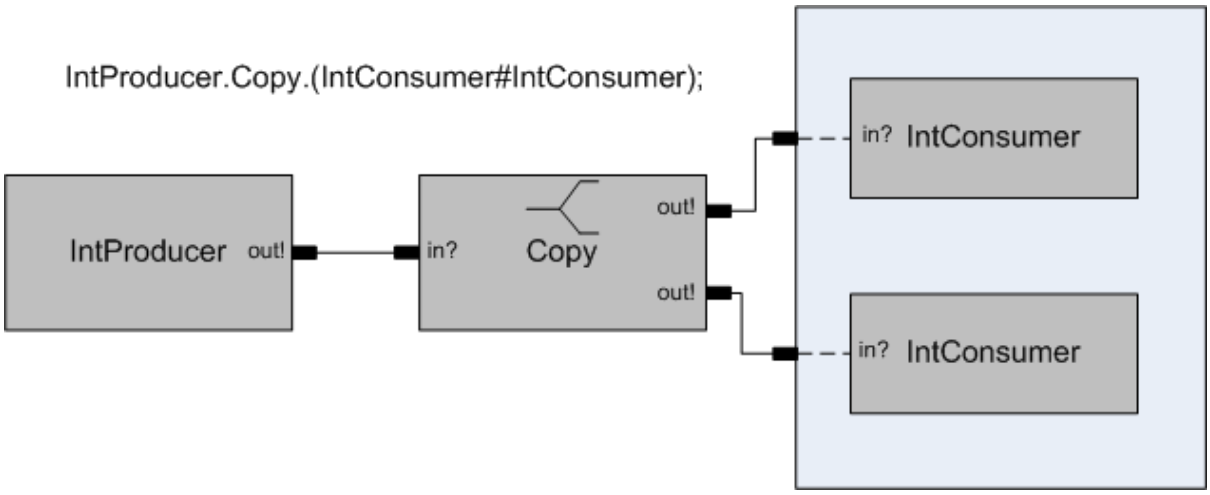


**Figure 31 - Unsupported feature of translator - use of special Copy component that encapsulates Copy synchronisation semantics. Currently, Copy only supported as an operator, '/\'.**

An endeavour was made to encapsulate a CopyWire object inside a component to achieve the required semantics. However, time constraints and problems encountered left this an unimplemented feature.

A temporary alternative approach uses a 'copy' *operator*, /\ (two slashes), similar to sequential composition, rather than a specially-defined Copy component:

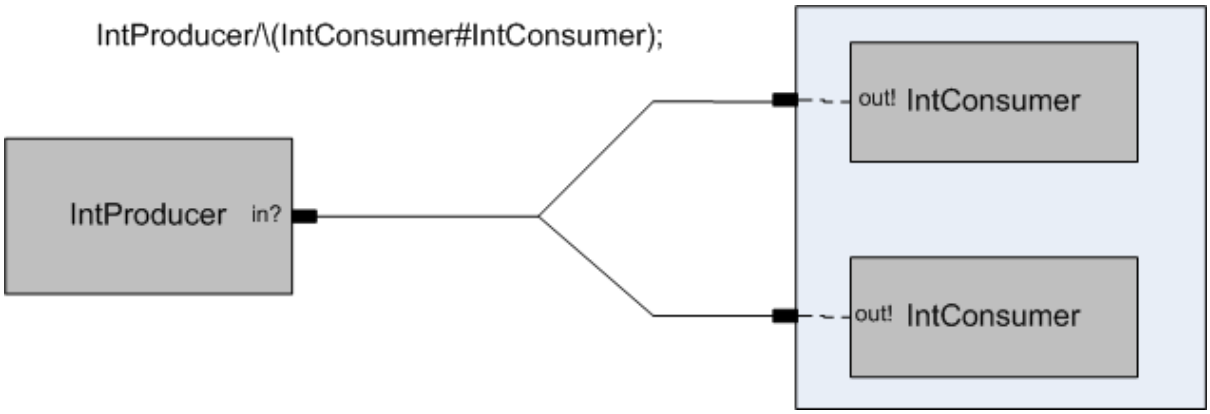


**Figure 32 - Alternative implementation of CopyWire into the translator, which allows a CopyWire to be placed between two compatible components (i.e. one boundary on left component, two boundaries on right component, same types and compatible directions) using the /\ operator.**

The drawback of this approach is that composability is restricted. The copy operator cannot be used in a tensor composition as the Copy component can:

(IntProducer#IntProducer).(Copy#IntConsumer)

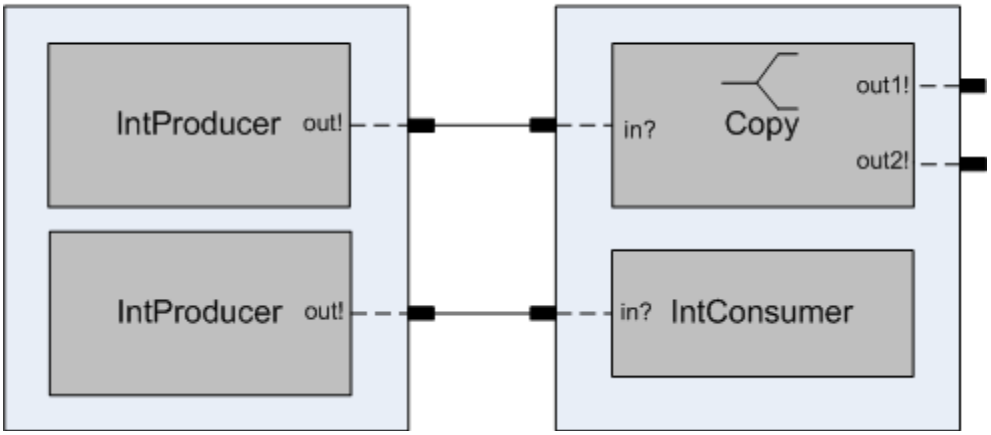


**Figure 33 - Tensor composition involving Copy *component*. Example of where the implemented Copy *operator* is insufficient.**

#### 4.1.3.6 Fairness

Currently, for both NormalWire and CopyWire, there is a potential fairness problem. Components can be starved of 'being-the-first-tugger' over multiple synchronisations. If a component arrives 'late' on the wire, it waits until the current synchronisation completes before it tugs again; however it could occur that the other component continually barges the late component, meaning that the late component never gets to tug first. Future work may require a queue to resolve this.

## 4.2 Translator High-Level Design

The phases of the translator are shown in Figure 34.

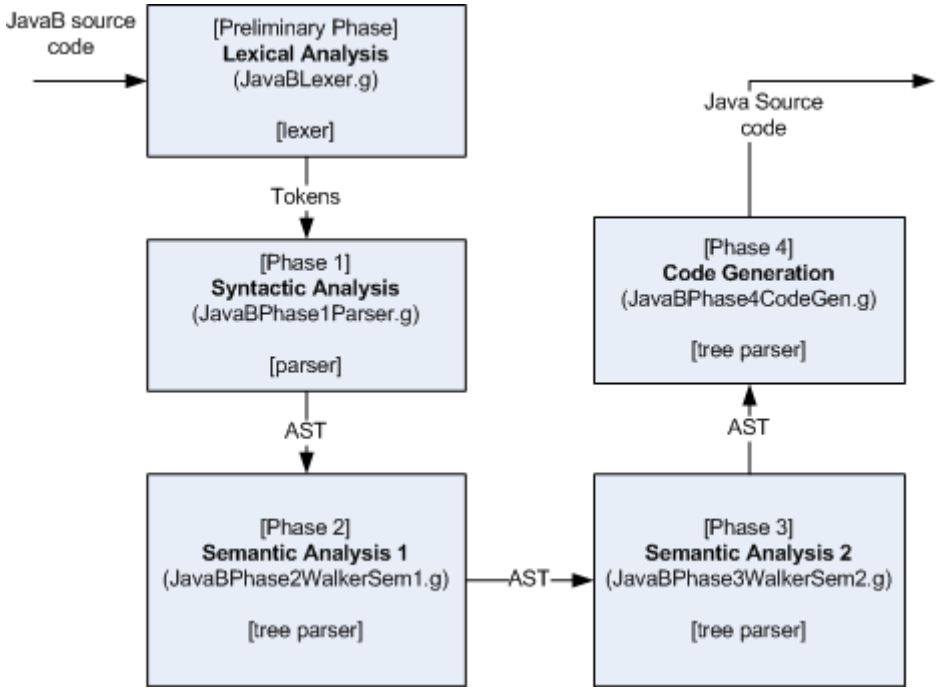


**Figure 34 - Translator Phases from input JavaB file to output Java file. ANTLR grammars are stored in .g files - the grammar file for each phase is indicated in brackets.**

JavaB source code is stored in .javab or .javabc files. .javab files contain wiring code. These contain any ordinary Java class/program with embedded JavaB wiring syntax within it. .javabc files contain component definitions, which may only contain a component definition at the top-level, though nested classes may appear within the component definition. Additionally, packages and imports may be still specified.

The translator performs syntactic and semantic checks of JavaB code, but only basic *syntactical* checks of Java code. Java code is passed through to the output verbatim. Thus, some (mainly semantic) Java errors can pass through the translator unnoticed. Consequently, javac must be used to check for such errors.

## 4.3 Translator Detailed Design and Implementation

A detailed explanation of each phase's workings follows. For each, only the *JavaB* rules in the grammar are explained (see attached DVD-ROM for full grammars). Where appropriate, the grammar-level options for each grammar are also explained.

Note that the grammars here are based upon the OpenJDK Compiler-Grammar project's Java grammar.

### 4.3.1 Unsuccessful Approaches

Despite some exposure to ANTLR previously, the requirements of the project necessitated some learning curve to fully appreciate ANTLR's capabilities and limitations. The result of this is that various approaches were taken to building the translator before an all-round effective solution was found. Figure 35 summarises this process:

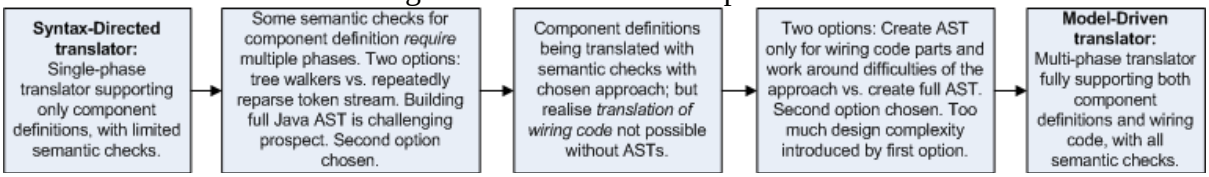


**Figure 35 - Approaches taken toward final design (transition from a syntax-directed translator to a model-driven translator)**

### 4.3.2 Lexical Analysis (JavaBLexer.g)

The lexer tokenizes the input character stream for the parser.

The only changes from the original Java grammar lexer was the addition of new lexer rules for new JavaB keywords (e.g. 'component', 'composition', 'boundary') and operators (e.g. '#', '/\').

```
// JavaB keywords/reserved words
COMPONENT   : 'component';
COMPOSITION : 'composition';
BOUNDARY    : 'boundary';
LEFT        : 'left';
RIGHT       : 'right';
RUN         : '__run__'; // run method keyword, was 'run'
BLOCK       : '__block__'; // block statement keyword, was 'block'
START       : '__start__'; // start statement keyword, was 'start'
// JavaB symbols
HASH        : '#'; // tensor composition operator
```

```
COPY        : '/\\'; // "copy wire" sequential composition temporary
*operator* -- note: backslash is escaped here
// Sequential Composition symbol . already used in Java lexer rules
```

**Figure 36 - Lexer rules for JavaB's keywords and symbols. Extract from JavaBLexer.g.**

Originally, 'run' and 'start' were used as keywords of the language. Due to their relatively commonality, particularly in multi-threaded programs, underscores were added to reduce potential conflicts.

Another point to note is that the ANTLR actions in the WS (whitespace), COMMENT, and LINE_COMMENT rules were changed from skip(); to $channel = HIDDEN;. This allowed the tokens representing comments to be hidden from the parser rather than be discarded completely, and thus ensured that whitespace and comments from the input were preserved in the generated output (in the code generation phase).

### 4.3.3 Syntactic Analysis (JavaBPhase1Parser.g)

The parser receives the tokens from the lexer and ensures they follow the language grammar. The parser matches the input tokens and builds up an AST as it goes.

```
parser grammar JavaBPhase1Parser;

options {
  language = Java;
  output = AST;
  backtrack = true; // backtracking required in original Java.g grammar
  memoize = true; // memoizing reduces time complexity (due to
backtracking), but increases space complexity
  tokenVocab=JavaBLexer;
}

/* Imaginary tokens (used as nodes of constructed AST) */
tokens {
  JAVAB_COMPILATION_UNIT;
  PACKAGE_DECL;
  MODIFIERS;

  ...
}

@header {
  import java.util.Map;
  import java.util.HashMap;
  import java.util.Set;
  import java.util.HashSet;
}

@members {
      // CONTEXT INSTANCE VARIABLES -- instance variables used for context
sensitivity (mainly used in gated semantic predicates) (see p125 of
hardcopy of ANTLR for example that has an inMethod instance variable)
      boolean inComponentDefinition = false;
      boolean inBoundaryDeclaration = false;
      boolean inHandlerDeclaration = false;
      boolean inRunMethodDeclaration = false;
      boolean inMethodDeclaration = false;

      // ERROR CHECKING (but no WARNINGS in this parser stage)
```

```
        private List<String> errorList = new ArrayList<String>();

        ...
}
```

**Code Listing 15 - Top of the parser grammar (JavaBPhase1Parser.g).**

The options show the fact that the output of this parser is an AST, backtracking is used (and memoizing) and the tokens to expect are the ones defined by JavaBLexer.g. The tokens section lists imaginary tokens used for nodes in the output AST[6]. The header section contains package and import statements that are included in the generated Java parser file. Likewise, the members section contains any field or method definitions to be included in the generated class. Here the members include variables that are used for keeping track of contextual information during the parse and also for storing error messages to be displayed at the end.

```
javaBCompilationUnit returns [List<String> returnErrorList]
        @init {
        $returnErrorList = this.errorList;
        }
    : ((annotations)? packageDeclaration)?
        (importDeclaration)*
        (componentDefinition | (typeDeclaration)*)
        -> ^(JAVAB_COMPILATION_UNIT annotations? packageDeclaration?
importDeclaration* componentDefinition? typeDeclaration*)
    ;
```

**Code Listing 16 - Start rule of parser grammar**

The start rule of the grammar shows that this error message list is returned at the end of the parse. It also shows how the imaginary token JAVAB_COMPILATION_UNIT is used as the root node of the subtree produced by this rule. The arrow denotes an *AST rewrite rule* that specifies the AST subtree to be constructed for that rule; the first element inside ^(...) is taken as the root node.

An important observation above is that a JavaB compilation unit may contain *either* a component definition *or* zero or more Java type declarations. *Wiring code* is permitted in type declarations (e.g. classes) but not in component definitions.

The following code listings show the *JavaB* rules of the grammar. Some Java rules were modified in order to integrate the JavaB rules. Only one is listed here. Note that references to lexer rules begin with upper case (e.g. COMPONENT), whereas references to parser rules begin with lower case (e.g. boundaryDeclaration).

```
// JAVA RULES REFERENCED: methodDeclaration, fieldDeclaration
componentDefinition
            scope {
            String componentName; // NOTE: handlerDeclaration rule needs
access to component name to pass to the template it invokes
            }
            @init {
            inComponentDefinition = true;
            }
            @after {
            inComponentDefinition = false;
```

[6] Imaginary tokens are tokens that do not have any input string associated with them but are used in the AST to represent psuedo-operations.

```
            }
    : COMPONENT IDENTIFIER LBRACE
      ( bds+=boundaryDeclaration
      | cfds+=fieldDeclaration
      | hds+=handlerDeclaration
      | rm+=runMethodDeclaration
      | mds+=methodDeclaration
      )*
      RBRACE
      -> ^(COMPONENT_DEF IDENTIFIER ^(BOUNDARY_DECLS ($bds)*) ^(FIELD_DECLS
($cfds)*) $rm* ^(HANDLER_DECLS ($hds)*) ^(METHOD_DECLS ($mds)*))
    ;
```

**Code Listing 17 - componentDefinition rule (in parser grammar)**

A component is defined by the component keyword, its name, and may contain boundary declarations, ordinary Java field declarations, handler declarations, a run method or ordinary Java method declarations. The semantic phase checks to ensure that a valid combination of boundaries, handlers and run methods have been provided; field and method declarations do not affect semantic validity of JavaB programs.

```
// JAVA RULES REFERENCED: type
// e.g. boundary left String bleftIn?;
boundaryDeclaration
            @init {
            inBoundaryDeclaration = true;
            }
            @after {
            inBoundaryDeclaration = false;
            }
    : BOUNDARY boundarySide type IDENTIFIER boundaryDirection SEMI
            -> ^(BOUNDARY_DECL IDENTIFIER boundarySide type
boundaryDirection)
    ;

boundarySide
    : LEFT
    | RIGHT
    ;

boundaryDirection
    : QUES
    | BANG
    ;
```

**Code Listing 18 - boundaryDeclaration and helper rules (in parser grammar)**

These boundary declaration rules are self-explanatory.

```
// JAVA RULES REFERENCED: blockStatement
runMethodDeclaration
            @init {
            inRunMethodDeclaration = true;
            }
            @after {
            inRunMethodDeclaration = false;
            }
    : RUN block
            -> ^(RUN_DECL[$RUN,"RUN_DECL"] block)
    ;
```

**Code Listing 19 - runMethodDeclaration rule (in parser grammar)**

A run method is simply the run keyword followed by an ordinary Java block (essentially statements inside curly braces).

```
// e.g. in?[int val] { code block }
// JAVA RULES REFERENCED: type, block
handlerDeclaration
            @init {
            inHandlerDeclaration = true; // used in gated semantic
predicates to provide turn alternatives on/off depending on whether we're
in a handler declaration context
            }
            @after {
            inHandlerDeclaration = false;
            }
    :   handlerName=IDENTIFIER boundaryDirection LBRACKET type
parameter=IDENTIFIER RBRACKET block
        -> ^(HANDLER_DECL $handlerName boundaryDirection type $parameter
handlerBlock)
    ;
```

**Code Listing 20 - handlerDeclaration rule (in parser grammar)**

Handler declarations share similar features to the previous two rules.

```
statement
    :   block
    |   ASSERT e1=expression (COLON e2=expression)? SEMI -> ^(ASSERT $e1
$e2?)
    |   IF parExpression s1=statement (options {k=1;}: (ELSE)=> ELSE
s2=statement)? -> ^(IF parExpression $s1 $s2?)
    |   forstatement
    |   WHILE parExpression statement -> ^(WHILE parExpression statement)
    |   DO statement WHILE parExpression SEMI -> ^(DO statement
parExpression)
    |   trystatement
    |   SWITCH parExpression LBRACE switchBlockStatementGroups RBRACE ->
^(SWITCH parExpression switchBlockStatementGroups)
    |   SYNCHRONIZED parExpression block -> ^(SYNCHRONIZED parExpression
block)
    |   RETURN (expression )? SEMI -> ^(RETURN expression?)
    |   THROW expression SEMI -> ^(THROW expression)
    |   BREAK (IDENTIFIER)? SEMI -> ^(BREAK IDENTIFIER?)
    |   CONTINUE (IDENTIFIER)? SEMI -> ^(CONTINUE IDENTIFIER?)
    |   expression SEMI -> ^(EXEC expression)
    |   IDENTIFIER COLON statement -> ^(LABELLED IDENTIFIER statement)
    |   SEMI -> SKIP
    // additional JavaB alternatives:
    |   {inComponentDefinition}?=> outSynchronizationStatement // can only
occur inside a component definition
    |   {inHandlerDeclaration}?=> handlerBlockStatement
    |   {!inComponentDefinition}?=> compositionDeclarationStatement // can
only occur in glue code
    |   {!inComponentDefinition}?=> startStatement // can only occur in
glue code
    ;
```

**Code Listing 21 - statement rule (in parser grammar)**

This Java rule has been augmented with four new alternatives, which use *gated semantic predicates* to enable/disable those alternatives during parsing depending on contextual information. For example, outward synchronisation statements (out![value];) are only permitted inside component definitions; never in wiring code. Additionally, 'block' statements are further restricted to the context of handlers. The listing below enumerates the component definition-only rules:

```
/* ********* COMPONENT DEFINITION ONLY CONSTRUCTS (only allowed within
component definitions, NOT glue code) ********* */
outSynchronizationStatement
    :   IDENTIFIER BANG LBRACKET expression RBRACKET SEMI
        -> ^(OUT_SYNC_STATEMENT IDENTIFIER expression)
    ;

inSynchronizationExpression
    :   IDENTIFIER QUES

    :   IDENTIFIER QUES
        -> ^(IN_SYNC_EXPR IDENTIFIER)
    ;

// e.g. block;
// only allowed within a component handler (only make sense inside
handlers)
handlerBlockStatement
    :   BLOCK SEMI
        -> BLOCK_STATEMENT
    ;
```

**Code Listing 22 - Rules representing 'component definition-only' constructs (in parser grammar)**

The following listing contains all the wiring code rules:

```
/* ********* COORDINATION/GLUE CODE ONLY CONSTRUCTS (only allowed within
wiring/glue code, NOT component definitions)********* */
compositionDeclarationStatement
    :   COMPOSITION IDENTIFIER EQ sequentialCompositionExpression SEMI
        -> ^(COMPOSITION_DECL[$COMPOSITION,"COMPOSITION_DECL"] ^(IDENT
IDENTIFIER) sequentialCompositionExpression)
    ;

/* COMPOSITION EXPRESSION HIERARCHY (only allowed within wiring/glue code
(they do not make sense inside component definitions)) */
/* Precedence is (highest to lowest):
 * 1. ID and parentheses (ID is a ref. to plain or composition component)
 * 2. Tensor composition
 * 3. Sequential composition / Copy sequential composition
 */

// NOTE: "copy wire" sequential composition is at equal precedence with
"normal wire" sequential composition
sequentialCompositionExpression
    : (tensorCompositionExpression -> tensorCompositionExpression) ((COPY
rightOperand=tensorCompositionExpression -> ^(COMPOSITION_COMPONENT ^(COPY
$sequentialCompositionExpression $rightOperand))) | (DOT
rightOperand=tensorCompositionExpression -> ^(COMPOSITION_COMPONENT ^(DOT
$sequentialCompositionExpression $rightOperand))))*
    ;

tensorCompositionExpression
```

```
    : (primaryCompositionExpression -> primaryCompositionExpression) (HASH
rightOperand=primaryCompositionExpression -> ^(COMPOSITION_COMPONENT ^(HASH
$tensorCompositionExpression $rightOperand)))*
    ;

primaryCompositionExpression
    :   compositionParExpression
    |   IDENTIFIER -> ^(PLAIN_OR_COMPOSITION_COMPONENT ^(IDENT IDENTIFIER))
    ;

compositionParExpression
    :   LPARAN sequentialCompositionExpression RPARAN
        -> sequentialCompositionExpression
    ;

/* END OF COMPOSITION EXPRESSION HIERARCHY */

startStatement
    :   START IDENTIFIER SEMI
        -> ^(START_STATEMENT ^(IDENT IDENTIFIER))
    ;
```

**Code Listing 23 - Rules representing 'wiring code-only' constructs (in parser grammar)**

The wiring code rules require some discussion. Given the following .javab file:

```
public class Application {
    public static void main(String[] args) {
        composition twoProdComp = TwoIntProducer.IntConsumer#IntConsumer;
        __start__ twoProdComp;
    }
}
```

**Code Listing 24 - Wiring code (.javab file) to create composition between TwoIntProducer and two tensored IntConsumers**

The *composition declaration statement*:

```
composition twoProdComp = TwoIntProducer.IntConsumer#IntConsumer;
```

assigns the right-hand *composition expression* to the composition twoProdComp.

Composition expressions are like ordinary expressions that require a precedence hierarchy for the different operators. In LL grammars [**52**] such as this one, the precedence of operators is encoded in the grammar rules by having the lower precedence rules invoke the higher precedence rules. Thus the order of precedence is (highest-to-lowest):

1. Reference to a plain or composition component (IDENTIFIER)
2. Tensor composition operator
3. Sequential composition operator and "copy" sequential composition operator

In the above example, the IntConsumer components are tensored before being sequentially composed with TwoIntProducer. As usual, parentheses may be used to override precedence. All the operators have left-to-right associativity.

Figure 37 shows the AST produced for the above composition declaration:

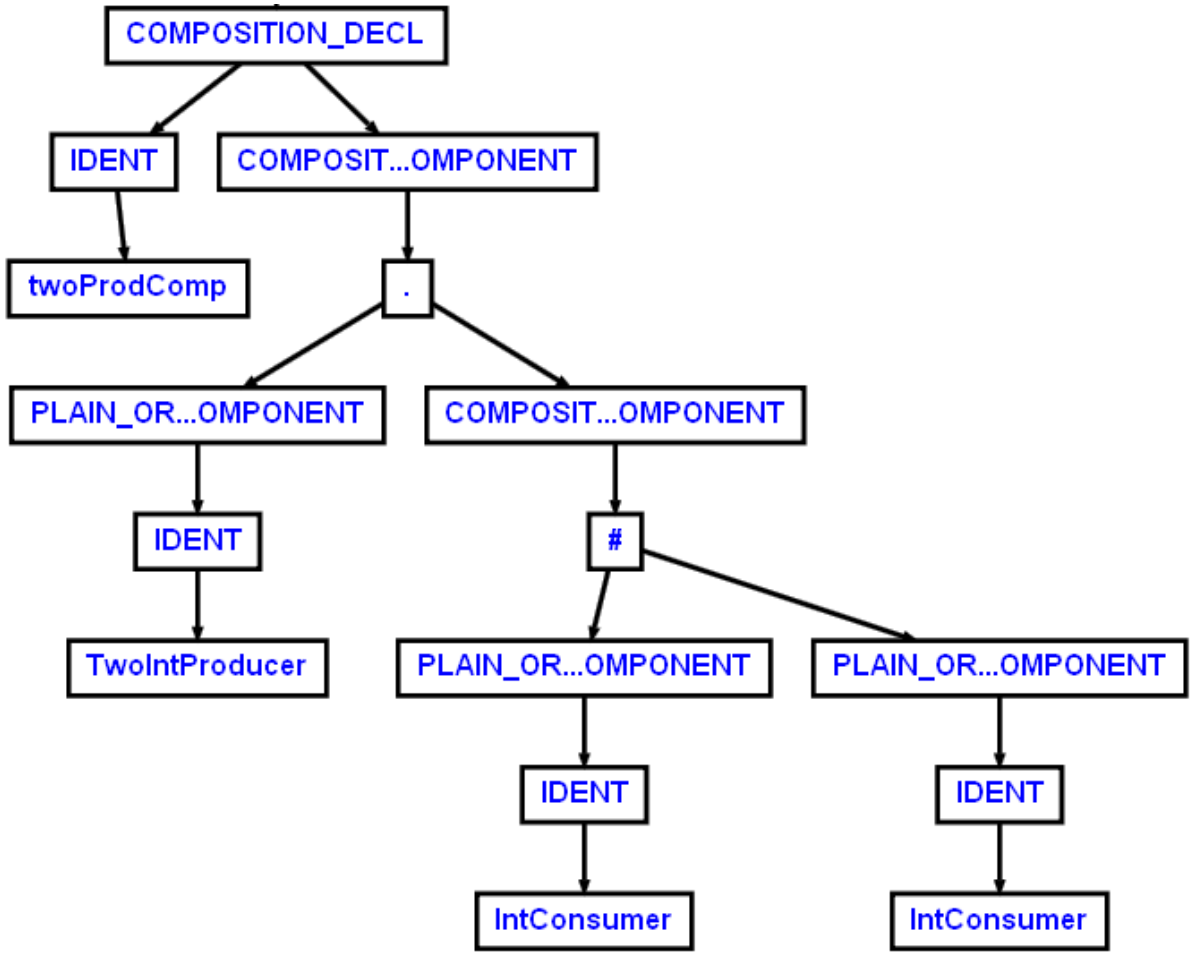


**Figure 37 - Composition declaration AST. This is not the AST for the entire program of Code Listing 24; it is only the subtree for the composition declaration.**

The ASTs produced for such composition declarations/expressions greatly simplify the following tree walker phases, in semantic analysis and code generation.

The final wiring rule, startStatement, is self-explanatory. It matches input such as:

```
__start__ twoProdComp;
```

where twoProdComp is a reference to a composition declaration.

#### 4.3.3.1 Example Abstract Syntax Tree (AST) Output

The ASTs for IntConsumer and the wiring code of Code Listing 24 above are now shown:

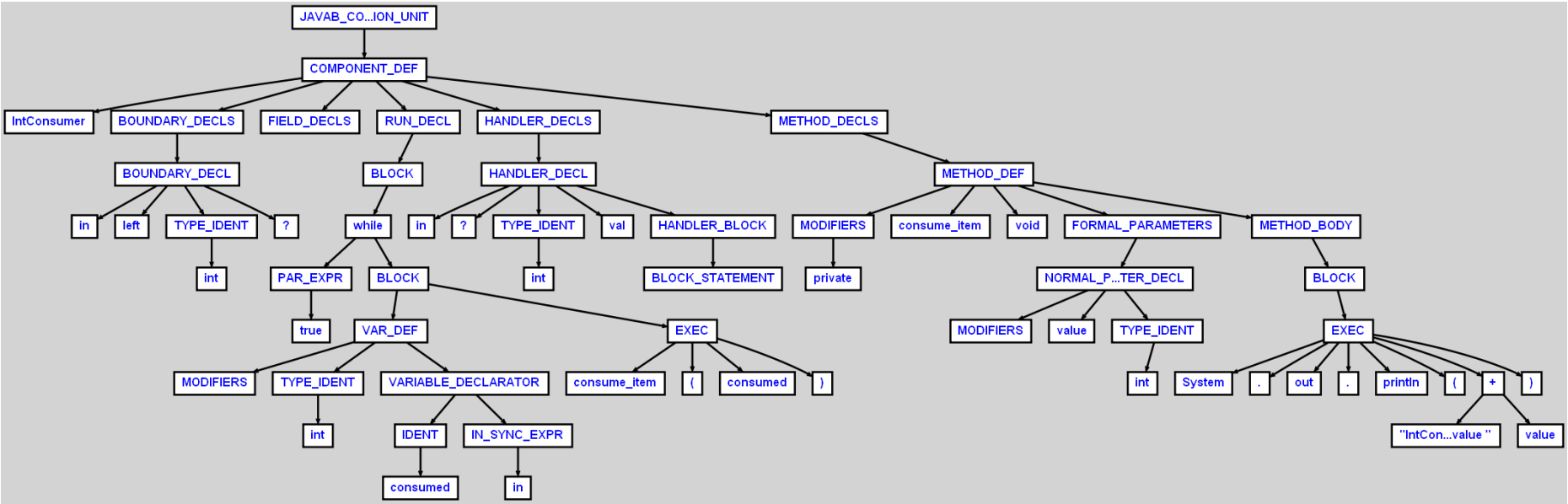


**Figure 38 - AST produced by parser for the IntConsumer component definition given in Code Listing 2**

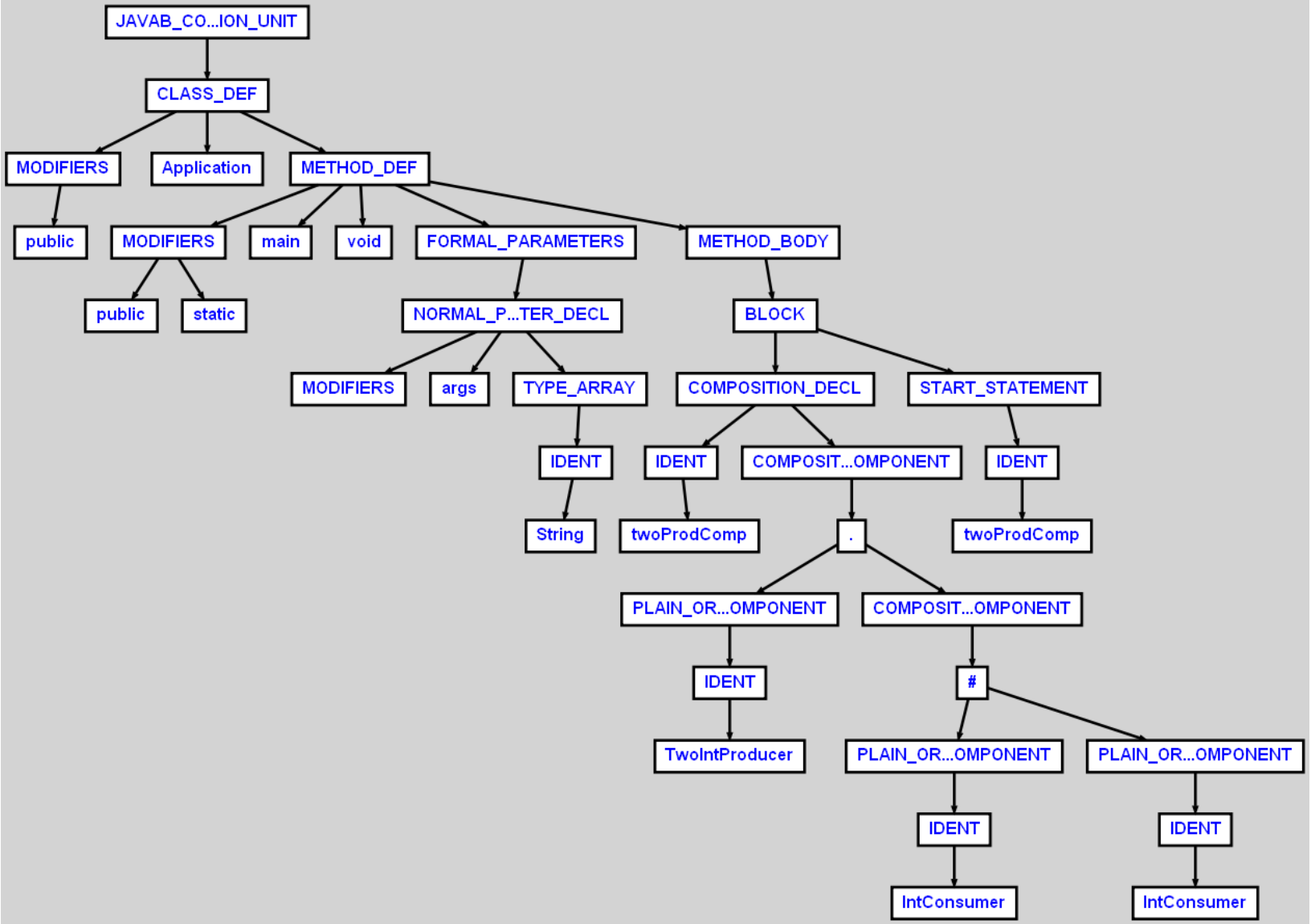


**Figure 39 - AST produced by parser for the wiring code in Code Listing 24**

### 4.3.4 Semantic Analysis (JavaBPhase2WalkerSem1.g and JavaBPhase3WalkerSem2.g)

The following two semantic phases perform a number of checks on the input program (in its condensed form, an AST). The majority involve looking up identifiers in symbol tables (e.g. component / boundary / composition symbol tables) and, for wiring code, ensuring a valid wiring has been given. Appendix G contains a full list of semantic checks.

All semantic checks for component definitions occur in the first semantic phase:

```
tree grammar JavaBPhase2WalkerSem1;

options {
  language = Java;
  output = AST;
  rewrite = true;
  backtrack = true;
  memoize = true;
  tokenVocab = JavaBPhase1Parser;
  ASTLabelType = CommonTree;
}

@header {
  import java.util.Map;
  import java.util.HashMap;
  import java.util.LinkedHashMap;
  import java.util.Set;
  import java.util.HashSet;
}

@members {
  // SYMBOL TABLES
  private Map<String,LinkedHashMap<String,Boundary>>
componentToLeftBoundariesSymTable;
  private Map<String,LinkedHashMap<String,Boundary>>
componentToRightBoundariesSymTable;

  private Map<String,CompositionDeclaration> compositionsSymTable;
  private Map<String,Boolean> componentToIsActive;

  // SEMANTIC CHECK INSTANCE VARIABLES (not the same as the context
instance variables used last time -- context was used in the parser to
determine if certain alternatives in certain rules were valid or not)
  private Set<String> handlerNames = new HashSet<String>();
  private int numRunMethods = 0;

  // SEMANTIC CHECK ERROR and WARNING lists
  private List<String> errorList = new ArrayList<String>();
  private List<String> warningList = new ArrayList<String>();

  /* Overridden */
  public void displayRecognitionError(String[] tokenNames,
RecognitionException e) {
    String hdr = getErrorHeader(e);
    String msg = getErrorMessage(e, tokenNames);
    errorList.add("ERROR: "+hdr + " " + msg);
  }
}
```

**Code Listing 25 - Top of phase 2 (semantic checks 1) tree grammar**

The important *options* here are output=AST and rewrite=true. The output option specifies that the output of this tree walker should be an AST. The rewrite option is a convenience option that allow the input AST (from the parser) to be copied to the output AST of the phase except where stated otherwise. Since most the time the AST does not need to be extensively modified, the rewrite option reduces the number of AST rewrite rules required to only where changes to the tree are needed.

Code Listing 25 also shows a number of symbol tables, implemented using java.util.Map, and variables, used for semantic checks. The two Maps `componentToLeftBoundariesSymTable` and `componentToRightBoundariesSymTable` map component identifiers to the boundaries of that component. The Boundary class here is used to represent a boundary, and is different from the Boundary class used by the generated translation (see section 4.1.1). The isCompatibleWith() method is used by the wiring semantic checks to ensure compatible boundaries are wired together:

```
public class Boundary {
	private String name;
	private String type;
	private Side side; // technically, don't actually need to store the
'side', since it is known implicitly by what Map the Boundary object is put
in
	private Direction direction;
	public Boundary(String name, String type, Side side, Direction
direction) {
		this.name = name;
		this.type = type;
		this.side = side;
		this.direction = direction;
	}

	// copy constructor (used in PlainComponent)
	public Boundary(Boundary b) {
		this.name = b.getName();
		this.type = b.getType();
		this.side = b.getSide();
		this.direction = b.getDirection();
	}

	public String getName() {
		return name;
	}

	public String getType() {
		return type;
	}

	public Side getSide() {
		return side;
	}

	public Direction getDirection() {
		return direction;
	}

	public String getDirectionString() {
		if(direction == Direction.IN) { return "?"; }
```

```
            else { return "!"; }
      }

      // is this boundary compatible with the given boundary? -- i.e.
compatible types, directions and sides
      public boolean isCompatibleWith(Boundary b) {
            boolean compatibleDirections = false;
            boolean compatibleSides = false;
            boolean compatibleTypes = this.type.equals(b.getType());

            // compatible directions?
            if(this.direction == Direction.IN) {
                  if(b.direction == Direction.OUT)
                        compatibleDirections = true;
            }
            else {
                  if(b.direction == Direction.IN)
                        compatibleDirections = true;
            }

            // compatible sides?
            if(this.side == Side.LEFT) {
                  if(b.side == Side.RIGHT) {
                        compatibleSides = true;
                  }
            }
            else { // this.side == Side.RIGHT
                  if(b.side == Side.LEFT) {
                        compatibleSides = true;
                  }
            }

            return (compatibleTypes && compatibleDirections &&
compatibleSides);
      }
}
```

**Code Listing 26 - Boundary class used to aid translation process. Represents a boundary of a component. Stores its name, type, side and direction. The most important method to note is the isCompatibleWith() method that may be used to check whether this Boundary is compatible with a given Boundary.**

Another important Map, `compositionsSymTable`, tracks all declared compositions in wiring code. All these symbol tables are passed from phase to phase, used for further semantic checks and also code generation.

Currently, one limitation of the translator is that in order to translate wiring code files, all component definition files it references must be translated at the same time. This is so that the translator can populate these Maps with the required information.

#### 4.3.4.1 Component Definition Semantic Checks

A small selection of semantic checks for *component definition* rules from JavaBPhase2WalkerSem1.g are now listed:

```
runMethodDeclaration
            @init {
            // update number of run methods seen (for semantic check
purposes)
            numRunMethods++;
```

```
            // update component -> isActive mapping of this component to
mark it as live since it has a run method

      componentToIsActive.put($componentDefinition::componentName,true);
            }
            :   ^(RUN_DECL block)
    {
    // SEMANTIC CHECK: enforce 0 (passive component) or 1 (active
component) run method
    if(numRunMethods > 1) {
      errorList.add("ERROR: ("+$RUN_DECL.line+":"+$RUN_DECL.pos+") Multiple
run methods defined. A component can define either zero (active components)
or one run method (passive components).");
    }
    }
    ;
```

**Code Listing 27 - There should only be zero or one run method declaration in a component definition**

```
boundaryDeclaration
          : ^(BOUNDARY_DECL IDENTIFIER boundarySide type boundaryDirection)
    {
    // SEMANTIC CHECK: no two boundaries with same identifier (i.e. ensure
this boundary identifier has not been used before). (Boundary identifiers
must be unique regardless of whether the rest of their signature is
different (i.e. their types or direction)).
    Side side = ($boundarySide.text.equals("left")) ? Side.LEFT :
Side.RIGHT;
    Direction direction = ($boundaryDirection.text.equals("!")) ?
Direction.OUT : Direction.IN;
    Boundary b = new Boundary($IDENTIFIER.text, $type.text, side,
direction);

    // check what existing boundaries there are for the current component
    if(side == Side.LEFT) {

if(!componentToLeftBoundariesSymTable.get($componentDefinition::componentNa
me).containsKey($IDENTIFIER.text)) { // .get will NOT return null -- we are
guaranteed that a component with that name exists
componentToLeftBoundariesSymTable.get($componentDefinition::componentName).
put($IDENTIFIER.text,b);
      }
      else { // error -- boundary already exists
        errorList.add("ERROR: ("+$IDENTIFIER.line+":"+$IDENTIFIER.pos+")
Boundary redeclaration. The boundary '"+b.getName()+"' has already been
declared in component '"+$componentDefinition::componentName+"'.");
      }
    }
    else if(side == Side.RIGHT) {

if(!componentToRightBoundariesSymTable.get($componentDefinition::componentN
ame).containsKey($IDENTIFIER.text)) {
        System.out.println("boundary "+$IDENTIFIER.text+" added to right
boundaries sym table");

componentToRightBoundariesSymTable.get($componentDefinition::componentName)
.put($IDENTIFIER.text,b);
      }
      else { // error -- boundary already exists
```

```
        errorList.add("ERROR: ("+$IDENTIFIER.line+":"+$IDENTIFIER.pos+")
Boundary redeclaration. The boundary '"+b.getName()+"' has already been
declared in component '"+$componentDefinition::componentName+"'.");
      }
    }
    }
    ;
```

**Code Listing 28 - Boundary declarations in a component definition are either added to the `componentTo[Left|Right]BoundariesSymTable` symbol table, or the boundary has been previously declared and an error is added to the list of errors.**

```
outSynchronizationStatement
    :   ^(OUT_SYNC_STATEMENT IDENTIFIER expression)
        {
        // SEMANTIC CHECK: identifier for boundary that we are sending on
actually exists
if(componentToLeftBoundariesSymTable.get($componentDefinition::componentNam
e).get($IDENTIFIER.text) == null &&
componentToRightBoundariesSymTable.get($componentDefinition::componentName)
.get($IDENTIFIER.text) == null) {
          errorList.add("ERROR: ("+$IDENTIFIER.line+":"+$IDENTIFIER.pos+")
Undeclared boundary '"+$IDENTIFIER.text+"' used in outward synchronization
expression.");
        }
        }
    ;
```

**Code Listing 29 - Outward synchronisations must take place on a boundary that exists in the current component (`$componentDefinition::componentName`).**

### 4.3.4.2 Wiring Code Semantic Checks

Wiring code checks primarily take place in the second semantic phase, in the compositionExpression rule. The alternatives for the rule that match different kinds of composition expression (e.g. sequential and tensor compositions) are listed separately and explained. The rule is recursive rule due to the nature of composition expressions, which may be nested indefinitely (see section 3.3).

```
compositionExpression returns [CompositionComponent compositionComponent]
// Sequential composition
// (e.g. IntProducer.IntConsumer)
    : ^(COMPOSITION_COMPONENT ^(DOT
leftCompositionComponent=compositionExpression
rightCompositionComponent=compositionExpression))
      // found composition component defined by a sequential composition
      {
      Map<String,Boundary> leftOperandRightBoundaries =
$leftCompositionComponent.compositionComponent.getRightBoundaries();
      Map<String,Boundary> rightOperandLeftBoundaries =
$rightCompositionComponent.compositionComponent.getLeftBoundaries();

      do {
            // SEMANTIC CHECK: also ensure that the wirings between these
boundaries are all compatible (where there is an incompatibility we need to
highlight it -- a mismatch in the number of boundaries is also an obvious
error -- check that first by comparing the sizes of lists....or something
similar)
               // if mismatch in size, then error
               if(leftOperandRightBoundaries.size() !=
rightOperandLeftBoundaries.size()) {
```

```
                    errorList.add("ERROR:
("+$leftCompositionComponent.start.getLine()+":"+$leftCompositionComponent.
start.getCharPositionInLine()+" to
"+$rightCompositionComponent.start.getLine()+":"+$rightCompositionComponent
.start.getCharPositionInLine()+")"+
                                    " Boundary mismatch in sequential
composition. The number of boundaries for the left operand and right
operand do not match.");
                    break;
                }

                // (at this point we know we have an equal number of
boundaries, however, they may not be compatible)
                // now check corresponding boundaries in the boundary lists
of the two operands are compatible
                Iterator<Map.Entry<String,Boundary>>
leftOperandRightBoundariesIterator =
leftOperandRightBoundaries.entrySet().iterator();
            Iterator<Map.Entry<String,Boundary>>
rightOperandLeftBoundariesIterator =
rightOperandLeftBoundaries.entrySet().iterator();
            while(leftOperandRightBoundariesIterator.hasNext()) {
                            Map.Entry<String,Boundary>
leftOperandBoundaryPair = leftOperandRightBoundariesIterator.next();
                            Boundary leftOperandBoundary =
leftOperandBoundaryPair.getValue();
                            if(leftOperandBoundary == null) { continue; } //
if the programmer actually referenced a non-existent boundary then this
will be null (the error associated with it will already have been added to
the error list)
                            Map.Entry<String,Boundary>
rightOperandBoundaryPair = rightOperandLeftBoundariesIterator.next();
                            Boundary rightOperandBoundary =
rightOperandBoundaryPair.getValue();
                            if(rightOperandBoundary == null) { continue; }

if(!leftOperandBoundary.isCompatibleWith(rightOperandBoundary)) {
                              errorList.add("ERROR:
("+$leftCompositionComponent.start.getLine()+":"+$leftCompositionComponent.
start.getCharPositionInLine()+" to
"+$rightCompositionComponent.start.getLine()+":"+$rightCompositionComponent
.start.getCharPositionInLine()+")"+
                                                " Incompatible boundaries in
boundary lists. Boundary '"+leftOperandBoundaryPair.getKey()+"' of left
operand is incompatible with corresponding boundary
'"+rightOperandBoundaryPair.getKey()+"' of right operand.");
                            }
                        }
                    } while(false);

      // construct appropriate object representing a sequential composition
of the two operands and their boundary lists (even if there is a semantic
error, this object will get instantiated anyway -- but that poses no
problem since the error will stop it going to the next phase: code
generation)
      $compositionComponent = new
SequentialCompositionComponent($leftCompositionComponent.compositionCompone
nt,$rightCompositionComponent.compositionComponent);
      }
```

**Code Listing 30 - The sequential composition (DOT) alternative of the compositionExpression rule in JavaBPhase3Sem2.g. Sequential composition requires the most important checks, since this is where wirings between boundaries occurs.**

The checks here ensure that for each pair of boundaries that are to be wired together, they are compatible (an error message is reported otherwise).

```
// "Copy" Sequential composition
// (e.g. IntProducer/\Cons#Cons)
    | ^(COMPOSITION_COMPONENT ^(COPY
leftCompositionComponent=compositionExpression
rightCompositionComponent=compositionExpression))
    {
      // semantic check of copywire wiring and then create
CopySequentialCompositionComponent object
      List<Map.Entry<String,Boundary>> leftOperandRightBoundaries = new
ArrayList<Map.Entry<String,Boundary>>($leftCompositionComponent.composition
Component.getRightBoundaries().entrySet()); // left operand should have a
single right boundary
      List<Map.Entry<String,Boundary>> rightOperandLeftBoundaries = new
ArrayList<Map.Entry<String,Boundary>>($rightCompositionComponent.compositio
nComponent.getLeftBoundaries().entrySet()); // right operand should have
two left boundaries

      do {
        // SEMANTIC CHECK: also ensure that the wirings between these
boundaries are all compatible (where there is an incompatibility we need to
highlight it -- a mismatch in the number of boundaries is also an obvious
error -- check that first by comparing the sizes of lists....or something
similar)
        // if the number of left and right boundaries is not compatible
with what a copy wire requires, then error
        if(leftOperandRightBoundaries.size() != 1) {
           errorList.add("ERROR:
("+$leftCompositionComponent.start.getLine()+":"+$leftCompositionComponent.
start.getCharPositionInLine()+")"+
                         " Boundary mismatch in COPY sequential
composition. Found "+leftOperandRightBoundaries.size()+" boundaries for
left operand, expecting just 1 boundary.");
           break;
        }
        else if(rightOperandLeftBoundaries.size() != 2) {
           errorList.add("ERROR:
("+$rightCompositionComponent.start.getLine()+":"+$rightCompositionComponen
t.start.getCharPositionInLine()+")"+
                         " Boundary mismatch in COPY sequential
composition. Found "+rightOperandLeftBoundaries.size()+" boundaries for
right operand, expecting just 2 boundaries.");
           break;
        }

        // (at this point we know we have the right number of boundaries
for the two operands, however, they may not be compatible)
        // now check corresponding boundaries in the boundary lists of the
two operands are compatible

                // get individual Map.Entry<String,Boundary> objects for
each boundary
                Map.Entry<String,Boundary> leftOperandBoundary =
leftOperandRightBoundaries.get(0);
```

```
                Map.Entry<String,Boundary> rightOperandTopBoundary =
rightOperandLeftBoundaries.get(0);
        Map.Entry<String,Boundary> rightOperandBottomBoundary =
rightOperandLeftBoundaries.get(1);


if(!leftOperandBoundary.getValue().isCompatibleWith(rightOperandTopBoundary
.getValue())) {
          errorList.add("ERROR:
("+$leftCompositionComponent.start.getLine()+":"+$leftCompositionComponent.
start.getCharPositionInLine()+" to
"+$rightCompositionComponent.start.getLine()+":"+$rightCompositionComponent
.start.getCharPositionInLine()+")"+
                          " Incompatible boundaries in COPY sequential
composition. Boundary '"+leftOperandBoundary.getKey()+"' of left operand is
incompatible with top boundary '"+rightOperandTopBoundary.getKey()+"' of
right operand.");
        }

if(!leftOperandBoundary.getValue().isCompatibleWith(rightOperandBottomBound
ary.getValue())) {
          errorList.add("ERROR:
("+$leftCompositionComponent.start.getLine()+":"+$leftCompositionComponent.
start.getCharPositionInLine()+" to
"+$rightCompositionComponent.start.getLine()+":"+$rightCompositionComponent
.start.getCharPositionInLine()+")"+
                          " Incompatible boundaries in COPY sequential
composition. Boundary '"+leftOperandBoundary.getKey()+"' of left operand is
incompatible with bottom boundary '"+rightOperandBottomBoundary.getKey()+"'
of right operand.");
        }
      } while(false);

      // construct appropriate object representing a COPY sequential
composition of the two operands and their boundaries (even if there is a
semantic error, this object will get instantiated anyway -- but that poses
no problem since the error will stop it going to the next phase: code
generation)
      $compositionComponent = new
CopySequentialCompositionComponent($leftCompositionComponent.compositionCom
ponent,$rightCompositionComponent.compositionComponent);
    }
```

**Code Listing 31 - The "Copy" sequential composition alternative of the compositionExpression rule in JavaBPhase3Sem2.g. (Recall that a Copy has been temporarily added to the translator as an operator rather than the original intention of a Copy component).**

The checks for Copy sequential composition are similar in nature to that of ordinary sequential composition. The only difference is that there must be *exactly* one sender boundary for its left operand component and *exactly* two boundaries for its right operand component.

```
// Tensor composition
// (e.g. IntConsumer#IntConsumer)
    | ^(COMPOSITION_COMPONENT ^(HASH
topCompositionComponent=compositionExpression
bottomCompositionComponent=compositionExpression))
          {
            // no semantic checks required for tensor composition
            $compositionComponent = new
TensorCompositionComponent($topCompositionComponent.compositionComponent,$b
ottomCompositionComponent.compositionComponent);
```

```
            }
```

**Code Listing 32 - The tensor composition component alternative of the compositionExpression rule in JavaBPhase3Sem2.g.**

Tensor does not require any semantic checks. It is not a *wiring* operator. Any component may be placed on any other component with no constraints.

```
// Reference to a plain / ordinary component
// (e.g. IntProducer)
    | ^(PLAIN_COMPONENT ^(IDENT IDENTIFIER)) // IDENTIFIER here refers to
the component *type* name of a plain component (not the instance name --
these are automatically generated internally by the code below)
      {
        System.out.println();
        System.out.println("Seen plain component: "+$IDENTIFIER.text);

        // SEMANTIC CHECK: Check the plain component exists
        // (no need to do this here since this was already checked in the
previous phase)
        // the check is:
if(componentToLeftBoundariesSymTable.get($IDENTIFIER.text) != null)


        // construct appropriate object representing plain component (in
particular, we are creating an INSTANCE of the component type)
        // Plain component constructor will automatically copy the Boundary
objects to ensure new Boundary objects are created (to ensure uniqueness of
Boundary objects)
        $compositionComponent = new PlainComponent($IDENTIFIER.text,
componentToLeftBoundariesSymTable.get($IDENTIFIER.text),
componentToRightBoundariesSymTable.get($IDENTIFIER.text));
      }
```

**Code Listing 33 - Alternative in the compositionExpression rule for when there is a reference to a plain component in JavaBPhase3Sem2.g.**

The requirement for references to plain components is that the component exists (been declared).

A previously declared composition may itself be referenced in a composition expression. For example:

```
composition c1 = IntProducer.IntBufferCell;
composition c2 = c1.IntConsumer; // c1 referenced
```

**Code Listing 34 - Referencing previously declared compositions within composition declarations.**

Its alternative in the compositionExpression rule follows:

```
// Reference to previously declared composition
// (e.g. c1 being referred to in composition c2)
    | ^(COMPOSITION_COMPONENT ^(IDENT IDENTIFIER)) // IDENTIFIER here
refers to a another declared composition
          {
          // SEMANTIC CHECK: reference to a composition component that has
actually been declared.
          // Lookup composition component in composition symbol table -- we
can assume it has already been declared, error if not found (reference to
an undeclared composition component)
          CompositionDeclaration declaredCompositionComponent =
compositionsSymTable.get($IDENTIFIER.text);
          if(declaredCompositionComponent != null) {
```

```
            $compositionComponent =
declaredCompositionComponent.getCompositionComponent();
          }
          else {
            errorList.add("ERROR:
("+$IDENTIFIER.line+":"+$IDENTIFIER.pos+")"+
                          " Reference to undeclared composition component
'"+$IDENTIFIER.text+"'.");
            // compositionComponent doesn't get set in the case of error;
doesn't matter however, because errors stop next phase processing it
          }
          }
    ;
```

**Code Listing 35 - Alternative in the compositionExpression rule for when there is a reference to a previously declared composition component.**

The requirement for references to composition components is that the composition component has been previously declared.

The reader may have observed that at the end of each of these alternatives, an object is constructed. These are summarised here:

```
// Sequential composition component is made up of its left and right
composition components
$compositionComponent = new
SequentialCompositionComponent($leftCompositionComponent.compositionCompone
nt,$rightCompositionComponent.compositionComponent);

// Copy sequential composition component is also made up of its left and
right composition components
$compositionComponent = new
CopySequentialCompositionComponent($leftCompositionComponent.compositionCom
ponent,$rightCompositionComponent.compositionComponent);

// Tensor composition component is made up of its top and bottom
composition components
$compositionComponent = new
TensorCompositionComponent($topCompositionComponent.compositionComponent,$b
ottomCompositionComponent.compositionComponent);

// Plain component is just itself; it is the atom / base case
$compositionComponent = new PlainComponent($IDENTIFIER.text,
componentToLeftBoundariesSymTable.get($IDENTIFIER.text),
componentToRightBoundariesSymTable.get($IDENTIFIER.text));

// Reference to previously declared composition component
$compositionComponent =
declaredCompositionComponent.getCompositionComponent();
```

**Code Listing 36 - Summary of the object instantiations that take place at the end of each alternative. These are not strictly part of the semantic phase but are preparation for the following code generation phase.**

`$compositionComponent` is the rule return parameter (see top of compositionExpression rule in Code Listing 30). Thus these objects are returned from the rule and a tree of ComponentComposition objects is built up as the AST is traversed. These objects' represent the composition components and are used in performing semantic checks of the sequential

composition alternative in particular (Code Listing 30). Secondly, they serve to simplify the code generation phase (see section 4.3.5.2).

## 4.3.5 Code Generation (JavaBPhase4WalkerGen.g and JavaBTemplates.stg)

For code generation, a template engine was required. The StringTemplate engine ([**53**] [**49**]) is well integrated with ANTLR and was the natural choice to use. Templates at their simplest contain placeholders for input parameters to be inserted whilst surrounding text is output verbatim. The templates themselves and how they are invoked from within an ANTLR grammar is presently discussed. The templates for generating component definition .java files are examined first, followed by the templates for wiring code .java files.

The code generation grammar is specified to have templates for its output rather than the usual AST:

```
tree grammar JavaBPhase4WalkerGen;

options {
  language = Java;
  output = template;
  rewrite = true;
  backtrack = true;
  memoize = true;
  tokenVocab = JavaBPhase3WalkerSem2;
  ASTLabelType = CommonTree;
}
```

**Code Listing 37 - Top of JavaBPhase4WalkerGen.g grammar. output = template is the key option to note. rewite = true is also important.**

Additionally, rewrite=true has been set. In this context (where output=template), this causes the underlying tokens associated with the input tree nodes to be rewritten to the output, except where there are template invocations from certain grammar rules that specify an alternative output (i.e. a translation). Thus, the Java rules in the ANTLR grammar are left untouched for the most part. Conversely, most JavaB rules specify a template that is used to translate into appropriate output.

### 4.3.5.1 Component Definition Templates

The componentDefinition template acts as the high-level template to which other template output is inserted. Its structure corresponds to the manual translation examples of section 4.1.1.

```
componentDefinition(name, boundaryDeclarations, fieldDeclarations,
handlerDeclarations, runMethodDeclaration, methodDeclarations) ::= <<
public class $name$ extends Component$if(runMethodDeclaration)$ implements
Runnable$endif$ {
      public $name$() {
            super("$name$"); // pass name of component to superclass
(Component)
      }
$if(fieldDeclarations)$

      // INTERNAL STATE
      $fieldDeclarations; separator="\n"$
$endif$
$if(boundaryDeclarations)$
```

```
      // BOUNDARIES
      $boundaryDeclarations; separator="\n"$
$endif$
$if(handlerDeclarations)$


      // HANDLERS
      $handlerDeclarations; separator="\n\n"$
$endif$
$if(runMethodDeclaration)$


      // RUN METHOD
      $runMethodDeclaration$
$endif$
$if(methodDeclarations)$


      // OTHER METHODS
      $methodDeclarations; separator="\n\n"$
$endif$

}
>>
```

**Code Listing 38 - componentDefinition *template* in JavaBTemplates.stg**

Dollar signs are used to delimit input parameters (as well as conditionals). The conditionals tests for the presence of the input parameter (null indicates absence). Code Listing 10 is an example of an output that follows the structure of a component definition shown here template.

The template is invoked in the ANTLR grammar by the correspondingly named rule, as seen below for componentDefinition. After the rule has matched its input, the template is invoked with values for each of its parameters.

```
componentDefinition
        scope {
        String componentName; // NOTE: handlerDeclaration rule needs access
to component name to pass to the template it invokes
        }
    :   ^(COMPONENT_DEF IDENTIFIER { $componentDefinition::componentName =
$IDENTIFIER.text; } ^(BOUNDARY_DECLS bds+=boundaryDeclaration*)
^(FIELD_DECLS fds+=fieldDeclaration*) rm+=runMethodDeclaration*
^(HANDLER_DECLS hds+=handlerDeclaration*) ^(METHOD_DECLS
mds+=methodDeclaration*))
        -> componentDefinition(name={$IDENTIFIER.text},
boundaryDeclarations={$bds}, fieldDeclarations={$fds},
handlerDeclarations={$hds}, runMethodDeclaration={$rm},
methodDeclarations={$mds})
    ;
```

**Code Listing 39 - componentDefinition *rule* in JavaBPhase4WalkerGen.g grammar, showing how the template is invoked. The arrow -> is followed by the invocation of the template with its input parameters.**

The following are a selection of templates whose output is eventually passed into the componentDefinition template by the componentDefinition rule above.

```
// component field declarations are the same as normal field declarations
except the only modifier allowed is the 'final' modifier (no public or
private keywords since everything is private by default)
// for component field declarations, we copy the input to the output for
the most part except that we add a 'private' keyword in front
componentFieldDecl(final, type, variableDeclarators) ::= <<
private$if(final)$ $final$$endif$ $type$ $variableDeclarators; separator =
", "$;
>>

// primitiveToReferenceTypesMap performs autoboxing of primitive types
boundaryDeclaration(boundaryName, type) ::= <<
private Boundary<$primitiveToReferenceTypesMap.(type)$> $boundaryName$;
>>

// the run method of a component definition; very simple translation
runMethodDeclaration(block) ::= <<
public void run() $block$
>>

// translation of handlers less trivial
handlerDeclaration(componentName, handlerBoundaryName, type, parameter,
handlerBlock) ::= <<
public Boundary<$primitiveToReferenceTypesMap.(type)$>
create_boundary_$handlerBoundaryName$(Wire<$primitiveToReferenceTypesMap.(t
ype)$> wireAttachedTo) {
      // the handler for this boundary

      HandlerRunnable<$primitiveToReferenceTypesMap.(type)$> handler = new
HandlerRunnable<$primitiveToReferenceTypesMap.(type)$>() {
            public $primitiveToReferenceTypesMap.(type)$
runHandler($primitiveToReferenceTypesMap.(type)$ $parameter$) {
                  // no translator housekeeping code required before user
code

                  // "user code" (with JavaB parts translated) -- which
could contain a (translated) 'block;' statement
                  $handlerBlock$

                  // translator housekeeping code following the user code
(if user code blocks then this code is unreachable)

      $handlerBoundaryName$.getWireAttachedTo().finishHandler($handlerBound
aryName$,$componentName$.this); // At this point we know that we have
finished the handler without blocking (i.e. the sync is complete, apart
from the housekeeping tasks we are about to do now)
                  return $parameter$;
            }
      };


      // create boundary (name, owner component, wire, handler)
      $handlerBoundaryName$ = new
Boundary<$primitiveToReferenceTypesMap.(type)$>("$handlerBoundaryName$",
this, wireAttachedTo, handler);
      return $handlerBoundaryName$;
}
```

```
>>
```

**Code Listing 40 - Selection of templates invoked by code generation phase grammar. The output of these templates eventually is passed as input into the componentDefinition template in Code Listing 38.**

One interesting feature to note above is `$primitiveToReferenceTypesMap.(type)$`, which is used to 'autobox' primitive types into their equivalent reference types. The translation mechanism uses on Java generics and thus autoboxing primitives is a necessary step.

The following rules illustrate the invocation of two templates from Code Listing 40:

```
outSynchronizationStatement
    :   ^(OUT_SYNC_STATEMENT IDENTIFIER expression)
        ->
outSynchronizationStatement(componentName={$componentDefinition::componentN
ame},boundaryToSendOn={$IDENTIFIER.text},exprValueToSend={$expression.text}
)
    ;

// specifically, only allowed within a component handler (only make sense
inside handlers)
handlerBlockStatement
    :   BLOCK_STATEMENT
        ->
handlerBlockStatement(componentName={$componentDefinition::componentName},h
andlerBoundaryName={$handlerDeclaration::handlerBoundaryName},handlerParame
ter={$handlerDeclaration::handlerParameter})
    ;
```

**Code Listing 41 - JavaB component definition rules in code generation grammar that invoke templates from Code Listing 40.**

As previously, it is ensured that all necessary input parameter are provided to the template.

### 4.3.5.2 Wiring Code Templates

In contrast to translating component definitions, only a single template is invoked to translate wiring code: startStatement. Thus a JavaB wiring program containing composition declarations that are *not* started would not yield any corresponding output in the generated Java file (the surrounding Java would still be output).

```
startStatement(componentInstancesToComponentTypeMap, normalWireWiringsList,
copyWireWiringsList, runnableComponentInstancesList) ::= <<
// create component instances contained in the composition
$componentInstancesToComponentTypeMap.keys:{instanceName |
$componentInstancesToComponentTypeMap.(instanceName)$ $instanceName$ = new
$componentInstancesToComponentTypeMap.(instanceName)$();$\n$}$

// create NormalWire and CopyWire instances
$normalWireWiringsList:normalWireInstantiation()$
$copyWireWiringsList:copyWireInstantiation()$

// create boundary objects
$normalWireWiringsList:createBoundaryConnectedToNormalWire()$
$copyWireWiringsList:createBoundaryConnectedToCopyWire()$

// now that we have created boundaries, set boundaries of the wire objects
$normalWireWiringsList:normalWireSetBoundaries()$
$copyWireWiringsList:copyWireSetBoundaries()$

/* Start threads of all live components (those that implement Runnable) */
```

```
// use a latch 'start gate' to ensure they start at the same time -- see
JCIP chapter 5
final CountDownLatch startGate = new CountDownLatch(1);

// add all Runnables to a set to be iterated over
Set<Runnable> runnables = new HashSet<Runnable>();
$runnableComponentInstancesList:{runnables.add($it$);$\n$}$

// set of latch-altered Runnables that have been turned into Threads
Set<Thread> threads = new HashSet<Thread>();

// iterate over them and wrap their run methods to include
startGate.await() at the beginning
for(final Runnable r : runnables) {
  Thread t = new Thread() {
    public void run() {
      try {
        startGate.await();
        r.run();
      }
      catch(InterruptedException e) { e.printStackTrace(); }
    }
  };
  threads.add(t);
  t.start(); // also start the thread (it will await at latch)
}

// GO! (release all the threads)
startGate.countDown();
>>
```

**Code Listing 42 - startStatement template, the only template invoked from the code generation grammar**

Its structure corresponds to the ProdConsApplication.java manual translation of section 4.1.2.2.

The invoking startStatement rule:

```
startStatement
    @init {
          // data structures to be passed to template (filled by recursive
algorithm)
          List<NormalWireBoundaryWiringAggregate> normalWireWirings = new
ArrayList<NormalWireBoundaryWiringAggregate>();
          List<CopyWireBoundaryWiringAggregate> copyWireWirings = new
ArrayList<CopyWireBoundaryWiringAggregate>();
          LinkedHashMap<String,String> instancesToComponentType = new
LinkedHashMap<String,String>(); // instance name -> component type name
          List<String> runnableInstances = new ArrayList<String>();
    }
    :   ^(START_STATEMENT ^(IDENT IDENTIFIER))
    {
          // ALL translation actually happens on the start statement --
composition declaration and expressions only fill symbol tables and build
up data structures etc. (mainly already done in previous phase)
          CompositionComponent compositionComponentToStart =
compositionsSymTable.get($IDENTIFIER.text).getCompositionComponent();

      // auxiliary data structures
      Map<String,Integer> componentTypeToNoOfInstances = new
HashMap<String,Integer>(); // component name -> no. of instances
```

```
      Map<Boundary,String> boundaryToOwningComponentInstance = new
HashMap<Boundary,String>();

          // TRAVERSAL -- invoke recursive algorithm to fill data
structures with information needed by the template
compositionComponentToStart.traverse(instancesToComponentType,componentType
ToNoOfInstances,boundaryToOwningComponentInstance,runnableInstances,compone
ntToIsLive,normalWireWirings,copyWireWirings);
    }
        // NOTE: we also pass all the runnables components so that all the
runnable components inside the composition are started (runnableInstances)
        ->
startStatement(componentInstancesToComponentTypeMap={instancesToComponentTy
pe},normalWireWiringsList={normalWireWirings},copyWireWiringsList={copyWire
Wirings},runnableComponentInstancesList={runnableInstances})
    ;
```

**Code Listing 43 - startStatement *rule* in code generation grammar which invokes startStatement *template*.**

The startStatement template itself is relatively simple. The complexity arises in deriving the values of its input parameters. The required parameters include:

- The names of component instances and their type. (`componentInstancesToComponentTypeMap`)
- The subset of component instances that are Runnable. (`runnableComponentInstancesList`)
- Information about all normal wirings (`normalWireWiringsList`)
- Information about all copy wirings (`copyWireWiringsList`)

As alluded to at the end of section 4.3.4.2, a hierarchy of objects is built up in the second semantic phase that represent a composition component. One purpose of this composition component object is that it simplifies the deriving of the input parameters above. The only required action of the startStatement rule (Code Listing 43) before invoking the template is to *traverse* the composition component that was started (compositionComponentToStart) to derive the input parameter values. This process is shown in Figure 40:

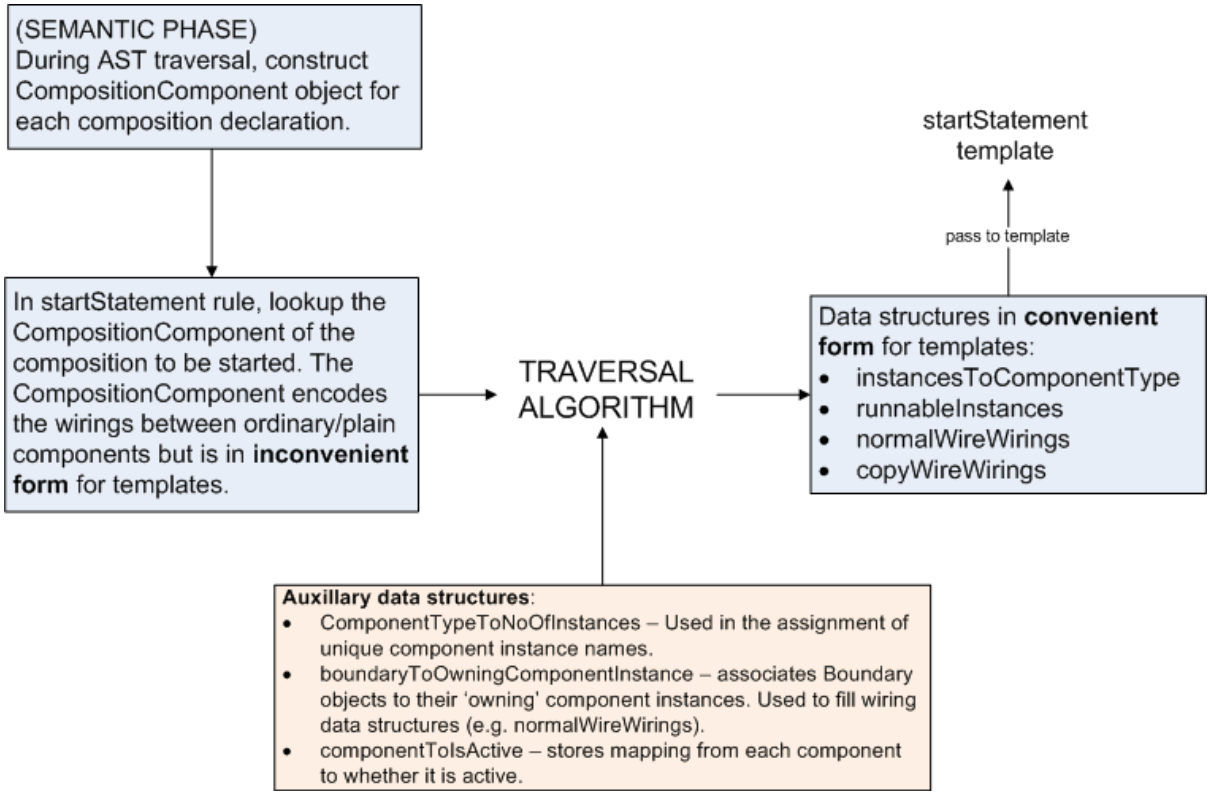


**Figure 40 - The traversal algorithm traverses the given CompositionComponent and fills the data structures to be passed to the startStatement template.**

A discussion of the traversal algorithm first requires an explanation of the `CompositionComponent` class hierarchy (Figure 41).

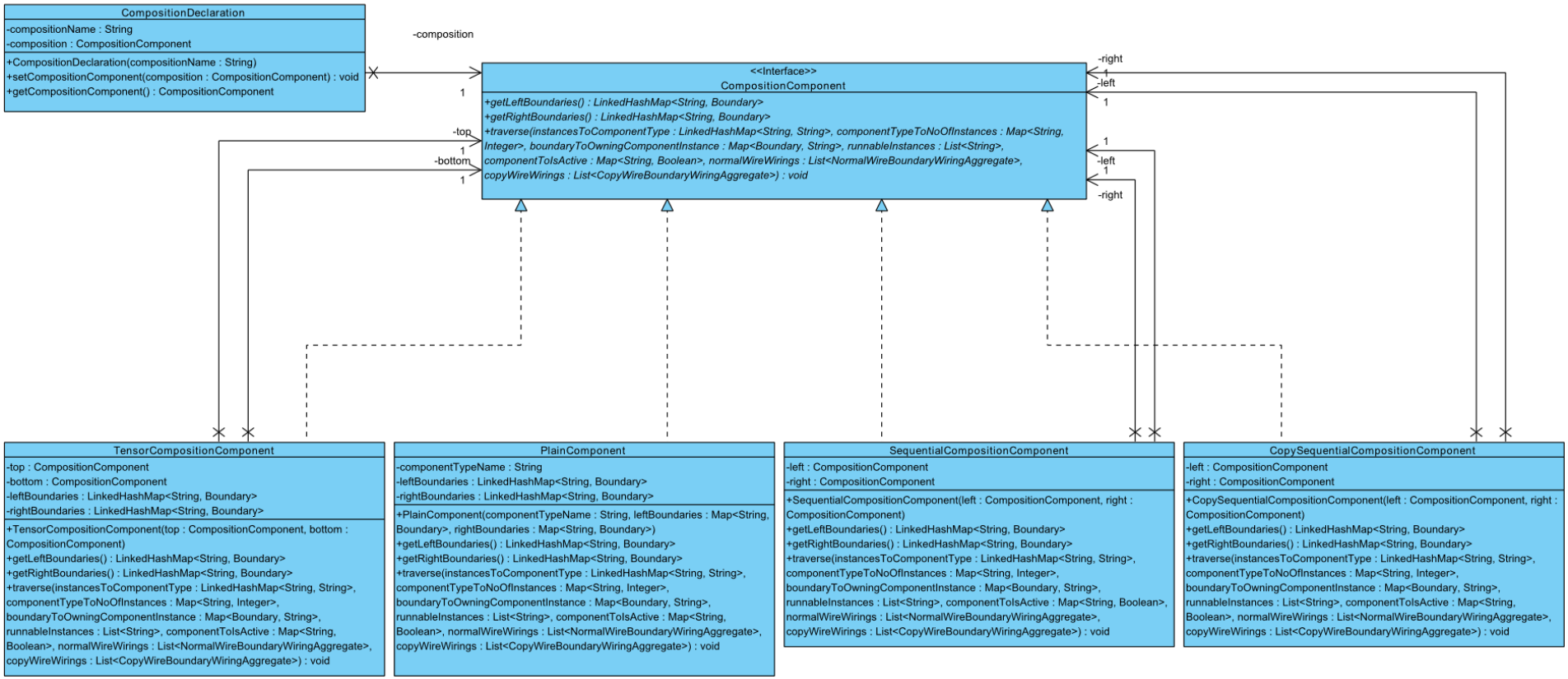


**Figure 41 - Class diagram of hierarchy between classes that each represent a component. This is an example of the Composite Pattern [54]. It can be seen that all implementing classes of CompositionComponent except PlainComponent have two references back to a CompositionComponent, one for each operand of the operation they represent (e.g. sequential composition, tensor composition, 'copy' sequential composition). These supplementary classes to the grammars may be found on the DVD-ROM.**

Both ordinary and composition components are modelled using these classes. `PlainComponent` represents an ordinary component (i.e. a component defined by a JavaB component definition). `SequentialCompositionComponent` represents the composition component resulting after the sequential composition operator has applied to its left and right operands. Similarly, `TensorCompositionComponent` represents the composition component resulting after the tensor composition operator has been applied to its top and bottom operands. The `CompositionComponent` interface represents any component (ordinary or composition), which the aforementioned classes implement. The operands of `SequentialCompositionComponent` and `TensorCompositionComponent` are themselves `CompositionComponent`s, as Figure 41 shows. As a result, *all `CompositionComponent` objects may be treated uniformly as components which have left and right boundaries.*

Every `CompositionComponent` class implements the traverse() method, though each implements it differently (polymorphism). Most implementations are recursive. Both `SequentialCompositionComponent` and `CopySequentialCompositionComponent` make recursive calls to traverse on their left and right operands. `TensorCompositionComponent` does likewise for its top and bottom operands. `PlainComponent` is the base case.

During the algorithm, `SequentialCompositionComponent` and `CopySequentialCompositionComponent` update the normalWireWirings and copyWireWirings lists, respectively. `TensorCompositionComponent` does not update any data structures (it simply makes the recursive calls). `PlainComponent` updates the instancesToComponentType and runnableInstances data structures.

### 4.3.6 Translator Controller code

This section briefly documents the main code of the translator that pulls all the phases together.

The translateSingleFile() method below shows clearly the relationship between the phases:

```
public boolean translateSingleFile(File f, boolean isGlueCodeFile) throws
IOException, RecognitionException {
  // phase 1 - syntactical analysis and produce AST (which may represent a
semantically incorrect program)
  ANTLRInputStream antlrInputStream = new ANTLRInputStream(new
FileInputStream(f));
  JavaBLexer lex = new JavaBLexer(antlrInputStream);
  TokenRewriteStream tokens = new TokenRewriteStream(lex);
  JavaBPhase1Parser phase1 = new JavaBPhase1Parser(tokens);
  JavaBPhase1Parser.javaBCompilationUnit_return r1 =
phase1.javaBCompilationUnit();
  // display errors (no warnings occur for parser stage (only errors))
  for(String error : r1.returnErrorList) { System.out.println(error); }
  // exit early if an error, else get the resulting AST
  if(r1.returnErrorList.size() > 0) { return false; }
  CommonTree t1 = (CommonTree)r1.getTree();
  generateStringAST(t1);
  generatePrettyAST(t1,f.getName()+"-ParserOutputAST");
  // end phase 1

  // phase 2 - walker for semantic analysis 1 (which collects info and
performs some of the semantic checks)
  CommonTreeNodeStream nodes1 = new CommonTreeNodeStream(t1);
  nodes1.setTokenStream(tokens);
```

```
  JavaBPhase2WalkerSem1 phase2 = new JavaBPhase2WalkerSem1(nodes1);
  JavaBPhase2WalkerSem1.javaBCompilationUnit_return r2 =
phase2.javaBCompilationUnit(componentToLeftBoundariesSymTable,componentToRi
ghtBoundariesSymTable,compositionsSymTable,componentToIsActive);
  // display errors and warnings
  for(String error : r2.returnErrorList) { System.out.println(error); }
  for(String warning : r2.returnWarningList) { System.out.println(warning);
}
  // exit early if an error (but not warnings -- warnings are not fatal)
  if(r2.returnErrorList.size() > 0) { return false; }
  // exit early if an error, else get the resulting AST
  if(r2.returnErrorList.size() > 0) { return false; }
  CommonTree t2 = (CommonTree)r2.getTree();
  generateStringAST(t2);
  generatePrettyAST(t2,f.getName()+"-Sem1OutputAST");
  // end phase 2

  // phase 3 - walker for semantic analysis 2 (rest of the semantic checks)
  CommonTreeNodeStream nodes2 = new CommonTreeNodeStream(t2);
  nodes1.setTokenStream(tokens);
  JavaBPhase3WalkerSem2 phase3 = new JavaBPhase3WalkerSem2(nodes2);
  JavaBPhase3WalkerSem2.javaBCompilationUnit_return r3 =
phase3.javaBCompilationUnit(componentToLeftBoundariesSymTable,componentToRi
ghtBoundariesSymTable,compositionsSymTable);
  // display errors and warnings
  for(String error : r3.returnErrorList) { System.out.println(error); }
  for(String warning : r3.returnWarningList) { System.out.println(warning);
}
  // exit early if an error (but not warnings -- warnings are not fatal)
  if(r3.returnErrorList.size() > 0) { return false; }
  // exit early if an error, else get the resulting AST
  if(r3.returnErrorList.size() > 0) { return false; }
  CommonTree t3 = (CommonTree)r3.getTree();
  generateStringAST(t3);
  generatePrettyAST(t3,f.getName()+"-Sem2OutputAST");
  // end phase 3

  // phase 4 - code generation (all errors assumed to be found by this
point)
  CommonTreeNodeStream nodes3 = new CommonTreeNodeStream(t3);
  nodes3.setTokenStream(tokens);
  JavaBPhase4WalkerGen phase4 = new JavaBPhase4WalkerGen(nodes3);
  phase4.setTemplateLib(templates); // give parser the templates

phase4.javaBCompilationUnit(compositionsSymTable,componentToIsActive,isGlue
CodeFile); // don't need return value because the token stream 'tokens' has
been rewritten

  File outputDirPath; // represents directory to file
  File outputFile; // the file itself
  try {
    String fileName = f.getName().substring(0,
f.getName().lastIndexOf("."))+".java"; // use same name as input file
(except the extension)
    System.out.println("Generated '"+ outputDir+fileName+"'");
    outputDirPath = new File(outputDir);
    outputDirPath.mkdirs(); // creates all necessary directories if they
don't already exist
    outputFile = new File(outputDir+fileName); // yes, that is meant to
include path too
    BufferedWriter out = new BufferedWriter(new FileWriter(outputFile));
```

```
    out.write(tokens.toString());
    out.close();
  } catch (IOException e) { e.printStackTrace(); return false; }
  // end phase 4

  // if we got to here in one piece, then translation succeeded
  lastGeneratedJavaFile = outputFile;
  return true;
}
```

**Code Listing 44 - translateSingleFile() method from JavaBTranslator.java. This is the core 'controller' code for translating a single JavaB file.**

## 4.4 Summary

This chapter discussed the core translation mechanisms, the algorithms that implement the required synchronisation, and finally the translator itself. With respect to the language specification laid out in Chapter 3, the core language features were successfully implemented. The Copy and Switch synchronisation primitives were not fully however.

# 5.

# Testing

This chapter documents the testing carried out on the translator and the translation mechanism classes described in chapter 4.

## 5.1 Testing the Translator

The use of gUnit [**55**] for testing the ANTLR grammars was considered. Though gUnit is a suitable tool, technical problems in running it rendered its use impossible. As an alternative, test case input programs were provided to the translator. The test cases are designed mainly to test JavaB constructs. This is because the Java rules (of the lexer and parser) do not need extensive testing because they are based on the Java grammar from the OpenJDK Compiler-Grammar project, which has already been extensively tested [**45**].

One known issue with the translator is that occasionally some ordinary Java code is not copied to the output translation as it should. This is due to shortcomings in the tree construction process. An example of this will be seen shortly.

### 5.1.1 Translator Test Cases

#### 5.1.1.1 Test Cases for JavaB Semantic Checks

Table 2 shows the test cases used for each semantic check expected of the translator. The tests used generally test only a single semantic check. It is possible that some bugs only surface under certain combinations of semantic checks. Such combinations were not tested.

| Test No. | Description | Input | Expected Output | Actual Output | Resolution |
|---|---|---|---|---|---|
| **Component Definitions** | | | | | |
| **General Semantic Checks** | | | | | |
| 1 | Names of components are distinct. No Two component may have same name. | Translator is passed three components, two components with same name. | Unique component translation succeeds. One of two components also succeeds but when translator reaches second one shows error message. | As expected. | N/A |
| 2 | Give *warning* if | Translator is | Translation | As expected. | N/A |

| Test No. | Description | Input | Expected Output | Actual Output | Resolution |
|---|---|---|---|---|---|
| | a component definition given which has neither run method nor any boundary declarations | passed a component as described. | succeeds but warning is shown. | | |
| 3 | Component has a declared boundary that has no corresponding handler. | Translator is passed a component as described. | Translator should automatically insert default handler that blocks into the generated output. | **Not as expected**. Translation 'succeeded' but only one handler in the generated Java code. There would be an error when compiling wiring code that uses this component. | This is a known feature still to be implemented. |
| **Boundary Declarations** | | | | | |
| 4 | No two boundaries of a component may have the same name, even if the rest of their signature is different | Translator is passed a component with three boundaries, two of which have the same name. | Translation fails with error that boundary has been re-declared. | As expected, except there was also spurious error messages regarding conflicting types of boundary declared and handlers (since two handlers declared with same name). | N/A (the spurious error messages cannot easily be removed). |
| **Run Method Declarations** | | | | | |
| 5 | No more than one run method in a component is permitted. | Translator is passed a component with two run methods. | Translation fails with error stating that there are multiple declared run methods. | As expected. | N/A |
| **Handler declarations** | | | | | |
| 6 | Handler has not already been declared. | Translator is passed a component that declares multiple handlers of same name. | Translation fails with error stating that handler has been re-declared. | **Not as expected**. Translation 'succeeded', with code generated for | Handler-Declaration rule in semantic1 grammar corrected. |

| Test No. | Description | Input | Expected Output | Actual Output | Resolution |
|---|---|---|---|---|---|
| | | | | both handlers (leading to two methods with same name)! | handlerNames, a Set that kept track of previously declared handlers was being updated in wrong branch of an if..else. |
| 7 | There exists a boundary with same name as the handler. | Translator is passed a component that declares a handler with no boundary of the same name. | Translation fails with error. | As expected. | N/A |
| 8 | Handler direction and type match that of boundary. (This test and the previous actually check that declared handlers have corresponding boundary declaration). | Translator is passed a component that declares a handler with same name as a declared boundary but incompatible direction and/or type. | Translation fails with error. | As expected. | N/A |
| **Synchronisation statements** | | | | | |
| 9 | For an inward / outward synchronisation statement, boundary being received / sent on actually exists. | Translator is passed a component with a run method containing an synchronisation statement on a non-existent boundary. | Translation fails with error. | As expected. | N/A |
| **Wiring Code** | | | | | |
| **Compositions** | | | | | |
| 10 | Composition is not redeclared with the same name as a previous composition declaration. | Translator is passed a wiring code application (along with required components) with two | Translation fails with error. | As expected. | N/A |

| Test No. | Description | Input | Expected Output | Actual Output | Resolution |
|---|---|---|---|---|---|
| | | compositions declared with the same name. | | | |
| 11 | Compositions must not reference an identifier that has not been previously declared (either a component or a composition). | Translator is passed a wiring application. Not all required components are passed as well. The wiring code's composition also references an undeclared composition. | Translation fails with error, stating that there is no component or composition with the name used in the composition. | As expected. | N/A |
| 12 | A sequential composition only wires compatible components. | Translator is passed a wiring application (along with required components). Wiring code tries to sequentially compose two components with incompatible boundaries. (IntConsumer was composed with IntProducer, rather than the other way round). | Translation fails with error, stating that the two components have incompatible boundaries. | Not as expected. Translation fails, but with (correct) error that there are 'dangling' boundaries remaining when attempting to start the composition. Reason for lack of error was that the test case components did not have *any* common boundaries between them at all, there was no error. | Alter translator to show error when two components which have *no common boundaries* are sequentially composed. This situation was not previously considered. |
| **Start Statements** | | | | | |
| 13 | Start statement must reference a declared composition. | Translator is passed a wiring application (along with required components). The wiring code's start statement references an undeclared composition. | Translation fails with error, stating that the start statement references an undeclared composition. | As expected. | N/A |

Testing

| Test No. | Description | Input | Expected Output | Actual Output | Resolution |
|---|---|---|---|---|---|
| 14 | When a composition is started, no 'dangling' / remaining boundaries are remaining that have not been wired to another boundary. | Translator is passed a wiring application (along with required components). The composition leaves some of its components with dangling boundaries. | Translation fails with error, stating that the composition cannot be started because it has dangling boundaries. | As expected. | N/A |

**Table 2 - JavaB semantic checks test cases. Based on the semantic checks of Appendix G.**

As can be seen, performing these tests revealed one or two omissions from the translator's semantic phases.

### 5.1.1.2 Example Program Test Cases

Various example programs were also run through the translator (some of which are given in Appendix B), as shown in Table 3.

| Test No. | Example Program | Translated Successfully | Compiled Successfully | Ran Successfully |
|---|---|---|---|---|
| 1a | P.C | Yes | No | - |
| 1b | P.C (with method call outside synchronisation statement; see below) | Yes | Yes | Yes |
| 2 | P.IBC.C | Yes | Yes | Yes |
| 3 | P.IBCx4.C | Yes | Yes | Yes |
| 4 | P.IBC.IBE | Yes | Yes | Yes |
| 5 | TwoIntProducer.(C#C) | Yes | Yes | Yes |
| 6 | P#P.C#C | Yes | Yes | Yes |
| 7 | SyncCounter.C | Yes | Yes | Yes |
| 8a | P. DiscerningIntConsumer | No | - | - |
| 8b | P. DiscerningIntConsumer (correction) | Yes | Yes | Yes |
| 9 | P.LazyIntConsumer | Yes | Yes | No |
| 10 | P/\C#C | Yes | Yes | No |
| 11 | P/\C#(IBC.C) | Yes | Yes | No |

**Table 3 - Example Program Test Cases. (P stands for IntProducer, C for IntConsumer, and IBC for IntBufferCell).**

The failure cases are now discussed.

*Test 1a*

The line:

```
out![produce_item()];
```

in IntProducer, was being translated to:

```
out.getWireAttachedTo().send(out,produce_item);
```

rather than the correct version:

```
out.getWireAttachedTo().send(out,produce_item());
```

The input Java is not copied to the translation output in its entirety. The exact cause of the bug was tracked down to unresolved issues in the Java tree construction process and their interaction with the outwardSynchronisationStatement rule in the code generation grammar.

The remaining examples deliberately avoid having such method calls directly inside outward synchronisation statements.

*Test 8a*

The failure occurred during parsing of the run method in DiscerningIntConsumer.javabc:

```
__run__ {
  int v; // <-- PARSER FAILS HERE
  while(true) {
    v = in?;
    while(v % 2 == 0)
      v = in?;
    consume_item(v);
  }
}
```

**Code Listing 45 - The run method of DiscerningIntConsumer.javabc causing a failure in the parser. This was due to a defect in the parser rather than an incorrect JavaB program.**

The specific errors were:

```
ERROR: line 8:2 mismatched input 'while' expecting RBRACE
ERROR: line 9:5 no viable alternative at input '='
```

**Figure 42 - Errors in Test 1a of Example Program Test Cases**

Correcting the multiplicity of the blockStatement rule invocation in the runMethodDeclaration rule resolved this:

```
runMethodDeclaration
    @init {
    inRunMethodDeclaration = true;
    }
    @after {
    inRunMethodDeclaration = false;
    }
    : RUN LBRACE blockStatement RBRACE // <-- CORRECTION: blockStatement*
    -> ^(RUN_DECL[$RUN,"RUN_DECL"] blockStatement)
    ;
```

**Code Listing 46 - Problem in runMethodDeclaration rule; correction to multiplicity of blockStatement**

Corresponding alterations to the tree grammars and the runMethod template were also required.

This bug had been concealed previously because most test cases used were non-terminating examples with a single while(true) blockStatement.

*Tests 9, 10, 11*

These tests all suffered from deadlock when the generated output was executed. Test 9 deadlocks due to the reasons documented in Appendix B.4. Tests 10 and 11 deadlock simply due to the fact that the current implementation of CopyWire contains unresolved deadlocks.

### 5.1.2 Supporting Classes

The main complexity in the translator's supporting classes was in the CompositionComponent classes (see section 4.3.5). JUnit was used to verify the correctness of the traverse() methods of these classes; that they fill the data structures correctly.

Two JUnit test case methods follow. The DVD-ROM contains the full test suite.

```
@Test
public void PlainTraverse() {
  // traverse the intProducer, passing (mostly) empty data structures
  intProd1.traverse(instancesToComponentType, componentTypeToNoOfInstances,
boundaryToOwningComponentInstance, runnableInstances, componentToIsActive,
normalWireWirings, copyWireWirings);

  // check results are correct
  assertEquals(instancesToComponentType.size(), 1);
assertTrue(instancesToComponentType.get("intProducer1").equals("IntProducer
"));
  assertEquals(componentTypeToNoOfInstances.size(),1);
  assertTrue(componentTypeToNoOfInstances.get("IntProducer") == 1);
  assertEquals(boundaryToOwningComponentInstance.size(),1);
  assertEquals(runnableInstances.size(),1);
  assertTrue(runnableInstances.contains("intProducer1"));

  // no wirings
  assertEquals(normalWireWirings.size(), 0);
  assertEquals(copyWireWirings.size(), 0);
}

@Test
public void SCCTraverseTwoPlainOperands() {
  // create seq comp. and traverse
  SequentialCompositionComponent scc = new
SequentialCompositionComponent(intProd1, intCons1);
  scc.traverse(instancesToComponentType, componentTypeToNoOfInstances,
boundaryToOwningComponentInstance, runnableInstances, componentToIsActive,
normalWireWirings, copyWireWirings);

  // check results are correct
  assertEquals(instancesToComponentType.size(), 2);
assertTrue(instancesToComponentType.get("intProducer1").equals("IntProducer
"));
assertTrue(instancesToComponentType.get("intConsumer1").equals("IntConsumer
"));
  assertEquals(componentTypeToNoOfInstances.size(),2);
  assertEquals(componentTypeToNoOfInstances.get("IntProducer"),new
Integer(1));
  assertEquals(componentTypeToNoOfInstances.get("IntConsumer"),new
Integer(1));
  assertTrue(boundaryToOwningComponentInstance.size() == 2);
  assertTrue(runnableInstances.size() == 2);
  assertTrue(runnableInstances.contains("intProducer1"));
  assertTrue(runnableInstances.contains("intConsumer1"));

  // a single wiring
  assertTrue(normalWireWirings.size() == 1);
  assertTrue(normalWireWirings.get(0).getBoundaryType().equals("int"));
assertTrue(normalWireWirings.get(0).getReceiverBoundaryName().equals("in"))
;
```

```
assertTrue(normalWireWirings.get(0).getReceiverInstanceName().equals("intCo
nsumer1"));
assertTrue(normalWireWirings.get(0).getSenderBoundaryName().equals("out"));
assertTrue(normalWireWirings.get(0).getSenderInstanceName().equals("intProd
ucer1"));

  // should be no copy wirings
  assertTrue(copyWireWirings.size() == 0);
}
```

**Code Listing 47 - Two JUnit test cases testing correctness of traverse() methods of PlainComponent and SequentialCompositionComponent**

## 5.2 Testing Translation Mechanism classes (inc. Wire)

The majority of the translation mechanism classes are trivial (see Appendix D), except the Wire classes. Thus only the testing of these is documented here.

### 5.2.1 NormalWire

An effective tool used for exposing concurrency bugs was ConTest [**33**]. The tool works by instrumenting Java bytecode with yields near synchronisation points. The deadlock discussed in section 4.1.3.1 was exposed using ConTest. Used in conjunction with ConTest was ECLEmma. This tool was used to analyse whether certain code paths in NormalWire were taken or not.

FindBugs was used as a supplementary aid to discovering bugs. It analyses the code for 'bug patterns'. Unfortunately, the only 'bugs' found were false positives.

### 5.2.2 CopyWire

Time constraints have meant testing of CopyWire thus far has also only used ConTest, and that only to expose deadlock.

In future work, the more complex semantics of CopyWire require more systematic testing by enumerating the different cases that can take place into equivalence classes. In particular, equivalence classes include the various orders components may tug (sender-receiver1-receiver2, receiver1-sender-receiver2 etc.). Additionally, the two ways handlers may complete also form two more equivalence classes, which when combined with the various orderings above, produce many more equivalence classes that would need to be tested. Performing this in conjunction with ConTest would be an effective strategy.

## 5.3 Summary

The semantic test cases and test programs successfully uncovered a number of bugs in the translator. For testing of the Wires, ConTest proved an invaluable tool.

# 6.

# Development Process and Tools

This chapter briefly discusses the development process chosen and software tools used in the project.

## 6.1 Process

The process adopted could be described as 'evolutionary iterative development'[**56**]. Evolutionary in the sense that the development of the language was relatively fluid and open to change. Often, language semantics were refined as implementation issues were encountered (e.g. development of NormalWire class and the translator itself clarified semantics of a wire and "sides" of boundaries, respectively). Iterative because the project moved from initial manual translations, to a core translator, to a translator that supported additional synchronisation primitives. Not all the planned iterations were completed however (see Figure 43 and Figure 44 in chapter 7). Overall, this process was the most natural approach for this project.

## 6.2 Tools

The software tools used in this project can be divided into several categories:

| Tool Type | Tools used |
|---|---|
| *Integrated Development Environment* | Eclipse used for development of 'manual translation' classes and other Java classes. The primary reasons for its use was familiarity and availability of plug-ins (some other tools below are actually Eclipse plug-ins e.g. ConTest, ANTLRIDE).<br><br>Netbeans/Ant were used to build the OpenJDK compiler when exploring approaches to translator construction [**46**]. |
| *ANTLR grammar development* | ANTLRWorks was initially used for grammar development. However, after numerous problems (bugs) with the tool, the Eclipse plug-in ANTLRIDE was used instead. Nevertheless, ANTLRWorks was still useful for performing (remote) debugging tasks.<br><br>ANTLRIDE also better automated the grammar build process. Unfortunately this build process was long. It built grammars that did not need rebuilding; this consistently added an extra 10 seconds to the development cycle of each build. |
| *Version Control System (VCS)* | The Mercurial VCS (with BitBucket[7] hosting) provided essential source control, backup and traceability. |
| *V&V* | ConTest (Eclipse plug-in), FindBugs, JUnit, JProfiler (see chapter 5) |
| *CASE Tools* | Visual Paradigm and Microsoft Visio were used for UML and generic diagrams seen in this report. |

**Table 4 - Software tools used during project**

[7] https://bitbucket.org/

# 7.

# Project Management

The main factor affecting the success of this project has been time. This chapter compares the project's goals and the plan in the progress report with the final outcome. It also documents some of the major problems that occurred.

## 7.1 Time Management

### 7.1.1 Overview

In general, the project's scope encompassed the following core activities:

1. Crystallising the language semantics
2. Implementing a manual translation (including NormalWire implementation)
3. Constructing the translator
4. Implementing further synchronisation primitives (Copy, Switch)

Activities one to three were indeed completed, whilst four was only partially. Thus overall, with respect to the goals laid out in the original project brief (Appendix A), the project has been relatively successful. The primary goal of developing the core language semantics and a translator was achieved.

Throughout the project, regular supervisor meetings helped keep the project on-focus.

### 7.1.2 Gantt Charts

Figure 43 shows the Gantt chart from the progress report (from mid-December). Figure 44, Figure 45, and Figure 46 show the final outcome Gantt chart in different formats (tasks, tasks with percentages, and summarised tasks with percentages, respectively). As well as indicating percentage progress, the *tasks* in the final Gantt chart are changed to reflect the true process that occurred. For example, for the 'core translator', the syntactic/semantic/generation subtasks are replaced with the translation of component definitions followed by translation of wiring code, which reflect the actual implementation order. Another difference is that work on the Copy construct took place *before* Switch. This approach was taken because it was felt that Copy would be the simpler one to implement.

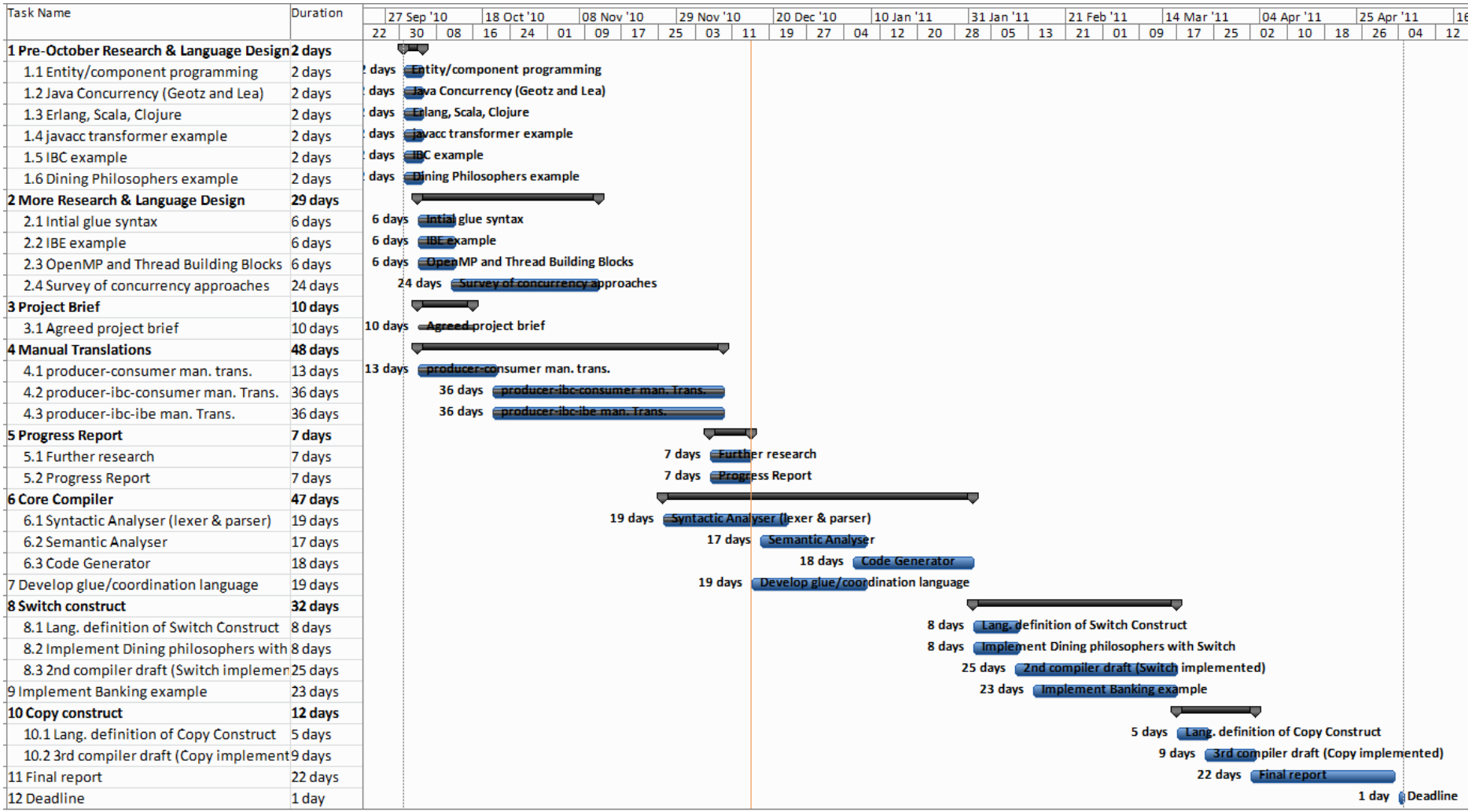


**Figure 43 - *Progress Report* Gantt chart of planned remaining work and expected order and/or parallelism of tasks.**

| Task Name | Duration | Start | Finish |
|---|---|---|---|
| **1 Pre-October Research & Language Design** | **2 days** | **Fri 01/10/10** | **Mon 04/10/10** |
| 1.1 Entity/component programming | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| 1.2 Java Concurrency (Geotz and Lea) | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| 1.3 Erlang, Scala, Clojure | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| 1.4 javacc transformer example | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| 1.5 IBC example | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| 1.6 Dining Philosophers example | 2 days | Fri 01/10/10 | Mon 04/10/10 |
| **2 More Research & Language Design** | **29 days** | **Mon 04/10/10** | **Thu 11/11/10** |
| 2.1 Intial glue syntax | 6 days | Mon 04/10/10 | Mon 11/10/10 |
| 2.2 IBE example | 6 days | Mon 04/10/10 | Mon 11/10/10 |
| 2.3 OpenMP and Thread Building Blocks | 6 days | Mon 04/10/10 | Mon 11/10/10 |
| 2.4 Survey of concurrency approaches | 24 days | Mon 11/10/10 | Thu 11/11/10 |
| **3 Project Brief** | **10 days** | **Mon 04/10/10** | **Fri 15/10/10** |
| 3.1 Agreed project brief | 10 days | Mon 04/10/10 | Fri 15/10/10 |
| **4 Manual Translations** | **48 days** | **Mon 04/10/10** | **Wed 08/12/10** |
| 4.1 producer-consumer | 13 days | Mon 04/10/10 | Wed 20/10/10 |
| 4.2 producer-ibc-consumer (NormalWire) | 36 days | Wed 20/10/10 | Wed 08/12/10 |
| 4.3 producer-ibc-ibe (NormalWire) | 36 days | Wed 20/10/10 | Wed 08/12/10 |
| **5 Progress Report** | **7 days** | **Mon 06/12/10** | **Tue 14/12/10** |
| 5.1 Further research | 7 days | Mon 06/12/10 | Tue 14/12/10 |
| 5.2 Progress Report | 7 days | Mon 06/12/10 | Tue 14/12/10 |
| 6 CW, Exams and Christmas | 32 days | Wed 15/12/10 | Thu 27/01/11 |
| **7 Core Translator** | **36 days** | **Thu 27/01/11** | **Thu 17/03/11** |
| 7.1 Translating Component Definitions | 24 days | Thu 27/01/11 | Tue 01/03/11 |
| 7.2 Translating Wiring Code (flexible boundaries) | 11 days | Tue 01/03/11 | Tue 15/03/11 |
| 7.3 Translating Wiring Code (fixed boundarie | 2 days | Tue 15/03/11 | Wed 16/03/11 |
| 8 Copy construct | 14 days | Tue 15/03/11 | Sat 02/04/11 |
| 8.1 Lang. definition of Copy Construct | 5 days | Tue 15/03/11 | Mon 21/03/11 |
| 8.2 Implement CopyWire (one sender directi | 12 days | Fri 18/03/11 | Sat 02/04/11 |
| 8.3 Implement Copy component (CopyWire internally) | 7 days | Fri 25/03/11 | Sat 02/04/11 |
| 8.4 2nd translator draft (Copy implemented) | 9 days | Wed 23/03/11 | Sat 02/04/11 |
| 9 Switch construct | 14 days | Tue 15/03/11 | Sat 02/04/11 |
| 9.1 Lang. definition of Switch Construct | 5 days | Tue 15/03/11 | Mon 21/03/11 |
| 9.2 Implement Dining philosophers with Swit | 5 days | Tue 15/03/11 | Mon 21/03/11 |
| 9.3 Implement Banking example | 5 days | Tue 15/03/11 | Mon 21/03/11 |
| 9.4 Implementation of SwitchWire | 12 days | Fri 18/03/11 | Sat 02/04/11 |
| 9.5 3rd compiler draft (Switch implemented) | 7 days | Fri 25/03/11 | Sat 02/04/11 |
| 10 Testing | 7 days | Sat 02/04/11 | Sat 09/04/11 |
| 11 Final report | 18 days | Sat 09/04/11 | Tue 03/05/11 |
| 12 Deadline | 1 day | Wed 04/05/11 | Wed 04/05/11 |

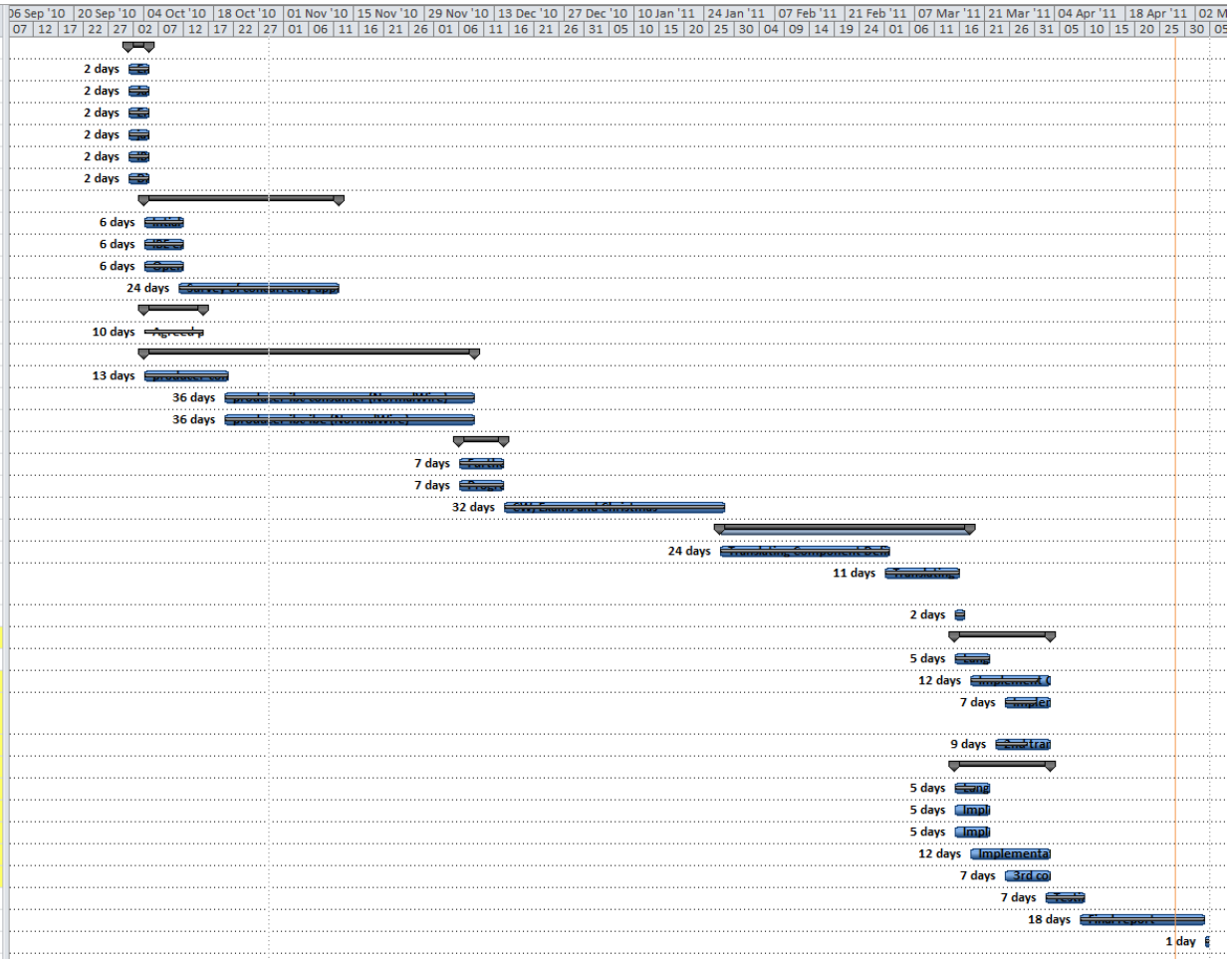


**Figure 44 - Gantt chart showing final outcome in terms of progress. Highlighted in yellow are all tasks that were either partially completed or not started. The Copy construct is an example of one that was partially completed.**

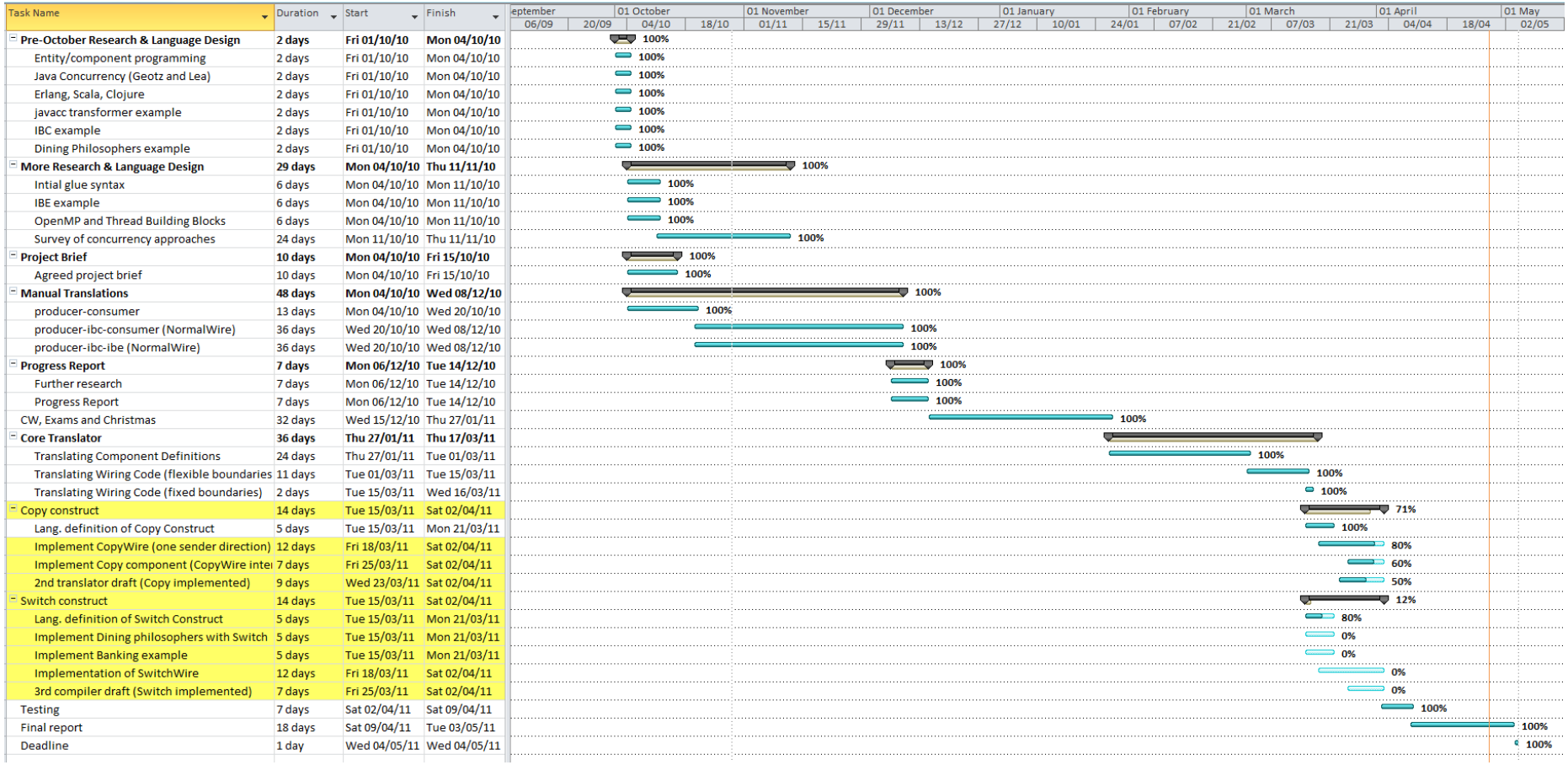


**Figure 45 - Identical to Gantt chart in Figure 44 but with percentages explicitly shown. Again, the partially completed tasks are shown in yellow.**

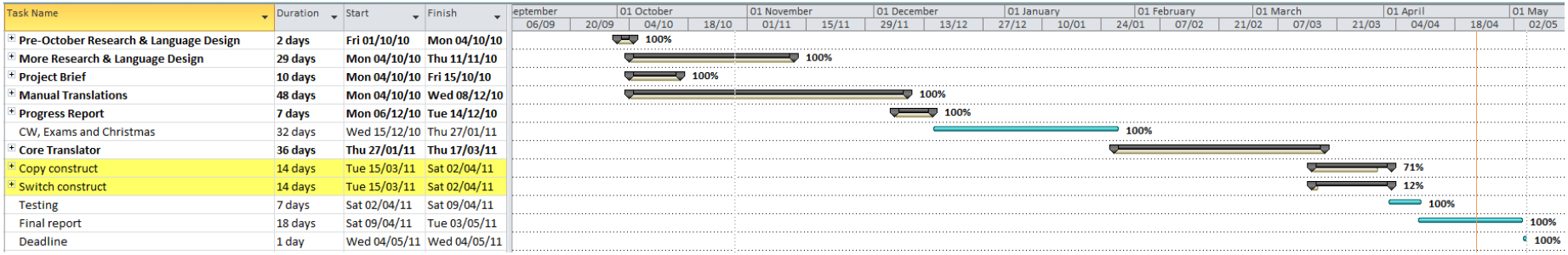


**Figure 46 - Summarised version of Figure 45 showing the overall tasks of the project and the progress made.**

### 7.1.3 Comparison of Forecast and Actual Progress

In the first term, there was no precise forecast for the project except that described in the project brief. This was partly due to the novel nature of the project. Most of the core semantics had already been discussed pre-term time. Additionally most of the manual translation was implemented within the first three weeks. However, difficulties encountered with NormalWire (see section 4.1.3.2), plus other coursework commitments, meant a full manual translation was only completed by the end of the first term.

Since the progress report, the primary focus has been translator implementation. The progress report Gantt chart predicted translator implementation by early February. This was an unrealistic goal however, since substantial work only began in late January. Coursework and exam commitments led to the decision to not focus on the project during the Christmas vacation and semester one exam period (see Figure 44). Nevertheless, the predicted time of six weeks to implement the translator was accurate.

During translator implementation, it was hoped it could have been finished sooner. However, as section 4.3.1 summarised, some approaches taken led to dead-ends, which necessarily required reworking and thus cost time. Inexperience in translator implementation (with ANTLR and StringTemplate) and the novel nature of the JavaB language were factors in this.

Delays in translator completion meant that the other synchronisation primitives could not be explored in full detail and/or implemented. The semantics of Copy were clarified and its implementation in one direction completed (although suffers from deadlock). The semantics and implementation of Switch had not begun.

## 7.2 Risk Management

The following table is based on that in the progress report. It enumerates the anticipated risks involved in the project but also notes actual occurrences of risks and how they were resolved.

| Risk (event) | Likelihood / Probability (1-5) | Impact / Loss incurred (1-5) | Risk Exposure (probability x loss) | Action (mitigation / avoidance / contingency plan) | Actual Occurrences of risk and how they were resolved |
|---|---|---|---|---|---|
| **Personnel difficulties** | | | | | |
| Supervisor is away | 4 | 1 | 4 | Keep in contact via e-mail | 1. Away in Paris when working on CopyWire and requiring a meeting. Resolved by rearranging meeting; this had low negative impact.<br><br>2. Away in Kenya during report write up period. Handled by e-mail. |
| Difficulty in communicating and getting along with supervisor | 2 | 3 | 6 | Try to resolve with supervisor. If unsuccessful try to resolve by seeing second examiner. | None. |
| Short-term illness (colds, flu etc.) | 3 | 2 | 6 | Get important work done well-before deadline; sleep enough | A cold during implementation of translator AST tree construction and tree grammar. Continued working but at slower pace. |
| Long-term illness/accident /injury | 1 | 5 | 5 | Contact supervisor to determine best course of action and fill in a Mitigating Circumstances Form (MCF) if necessary. | None. |
| Supervisor has short-term illness | 3 | 1 | 3 | Keep in contact via e-mail (if necessary) | Once. But of no significant impact. |
| Supervisor has long-term illness / | 1 | 4 | 4 | May be able to keep in contact via e-mail (or | None. |

| | | | | | |
|---|---|---|---|---|---|
| accident / injury | | | | phone).<br><br>This project is highly reliant on the well-being of my supervisor so a "replacement supervisor" might not be sufficient! | |
| **Technical difficulties** | | | | | |
| Lack of progress in language development | 3 | 4 | 12 | Consult supervisor. | Some challenges in determining semantics of some constructs but these were overcome. |
| Difficulty in building translator | 3 | 3 | 9 | Consult supervisor. Possibly invest more time into compiler construction to just get a working first version compiler only.<br><br>Read compiler books to gain inspiration/ideas. | Many difficulties, reflected in the number of dead-end approaches taken (summarised in section Unsuccessful Approaches4.3.1). Many discussions with supervisor. [**48**] was a core reference aid. However[**49**] provided significant help in using StringTemplate. A little more than first version was completed. |
| **Project/Schedule difficulties** | | | | | |
| A minor fall behind schedule (<= 3 weeks) | 5 | 1 | 5 | Take time to rethink through tasks and schedule. Re-plan time. There is still time to recuperate. | Implementation difficulties made this a common occurrence. Plans were changed and also talked through with supervisor. |
| A major fall behind schedule (> 3 weeks) | 3 | 4 | 12 | Talk with supervisor about what to do, what tasks to not spend time on, and generally what the best course of action is. | It was realised that Copy and Switch could not both be implemented. Thus Switch was dropped. |

| | | | | | |
|---|---|---|---|---|---|
| Coursework of other modules is difficult / overwhelming | 1 | 4 | 4 | Only courseworks are 30% *COMP3011 Critical Systems* and 30% *COMP3006: Real-time Computing and Embedded Systems*.<br><br>Mitigate effect by planning my time early, limiting amount of time spent on such coursework to get 80% of the marks and not spend time on the last 20%. | Small impact in second semester.<br><br>However, in first semester this had large impact. In particular, COMP3004 coursework meant little work on translator was achieved over Christmas period. |
| Both computer and backup hard drive fail | 1 | 5 | 5 | Regular backup to ECS servers. Backup to another backup hard drive and another computer. Push changesets to revision control system (Mercurial) frequently! | None. |

**Table 5 - Risk assessment based on that in progress report, including actual occurrences and how they were resolved**

The primary risks that had occurred that had the greatest impact were: coursework from other modules competing with the project and also difficulties in implementing the translator.

## 7.3 Summary

Overall, despite encountering some significant problems during the project, these were not show-stopping. A good level of progress was achieved and most project goals were met.

# 8.

# Conclusions and Future Work

This chapter concludes by evaluating the project's achievements with respect to the goals set. It then discusses possible future work on the JavaB language.

## 8.1 Conclusions

The vision expressed in the introduction was to develop a language that simplifies the construction of concurrent software and makes it possible to verify their correctness. This project has made initial inroads into fulfilling these aspirations.

The project has focused on development of a language that extends Java - JavaB. JavaB's core semantics were clarified. This included the concepts of components, boundaries, handlers and wires. Additionally, the core operators, sequential composition and tensor composition were defined. The semantics and implementation of JavaB's synchronisation primitives was well underway, with Copy (but not Switch) mainly implemented. An initial translator that converts JavaB constructs into Java's lower-level concurrency primitives was completed. The translator's model-driven architecture with separate phases (using ASTs) makes extending the translator very easy.

The ambition of composing components to achieve sophisticated synchronisations has been partially achieved. The operators of the language allow components to be composed. Indeed, some interesting combinations of components can be constructed (see section 3.3). However, much of the power of composability cannot be realised without standard components. These include the synchronisation primitive components such as Copy and Switch, and also components which enable flexible wirings such as Twist, Identity and IdentityLoopback (see Appendix C).

The focus on implementation meant limited time could be given to the application of JavaB to real-world applications.

## 8.2 Suggestions for Future Work

There is plenty of scope for extending this project.

*Development of Existing Work*
Firstly, there are a number of possible improvements to the translator. The AST construction process for the Java rules require some corrections. Occasionally, some Java input is 'lost' in the generated output. An alternative Java grammar by Dieter[8], which includes tree construction operations, could be used to aid this process (or even used in place of the

[8] http://www.antlr.org/grammar/1207932239307/Java1_5Grammars

existing grammars). Another improvement would be better error messages. The semantic error messages are good; however, syntactical error messages are not very user-friendly, mainly due to the use of backtracking. Additionally, it would be desirable for the translator to convert directly to bytecode without requiring javac. Modifying the OpenJDK javac compiler would likely be a more sensible approach than implementing this by modifying the existing translator.

The existing Wires also have scope for improvement. Correcting CopyWire to avoid deadlock, implementing SwitchWire and encapsulating both as standard components are essential work. Implementing the other standard components of Appendix C is also a key improvement. Model checking tools such as Java PathFinder could be used to verify the correctness of the Wire algorithms. Finally, fairness (see section 4.1.3.6) and performance of the wires are further considerations.

*Research Directions*

One interesting idea that could be explored is allowing multiple boundaries to be treated as a single 'logical' boundary. A synchronisation on such a logical boundary behaves like an ordinary 'single-boundary' synchronisation. For example a component could send integers on two boundaries o1 and o2 using the following synchronisation statement:

```
(o1,o2)![5,6];
```

Such a synchronisation would only complete once both 'sub-synchronisations' complete for the tugs that take place on o1 and o2. The owning component of these boundaries defines a handler for the logical boundary.

An example of this could be an IntProducer that only sends out two integers at a time.

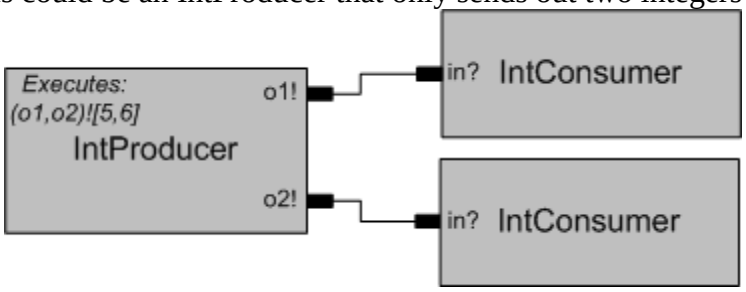


**Figure 47 - A modified IntProducer to previously seen, that treats o1 and o2 as one logical boundary. The IntConsumers are the same as previously seen.**

It is conceivable that there applications where there are components that want to synchronise with (i.e. be in step with) with several other components simultaneously, in a single transaction-like fashion.

Session types [**57**] [**58**] represent another research direction. These provide a *higher-level type system* that not only describe simple types but actually *describe a protocol*. Thus session types could be used on wires to describe a protocol which the communications on the wire must follow. For example, a very simple protocol could be that there are always two sends followed by a receive.

They may also be used to attach predicates to synchronisations that happen on a wire. For example, two boundaries sending integers could have a predicate that specifies that the two integers must sum to 10.

The type checking for session types would occur at both compile-time and run-time. Many checks can actually be performed at compile-time.

Finally, further directions that could be explored include:

- Dynamic boundaries, where boundaries may change at runtime
- Subtyping of components in a similar way to that in Object-Oriented languages. Components may inherit boundary declarations, handlers, fields and methods.
- Rolling back of synchronisations in a similar way to database transactions
- Developing methods to formally verify the correctness of JavaB programs
- Developing a graphical environment for designing networks of components

## 8.3 Summary

This has indeed been a challenging project. Much was learned through the process, both in technical skills (e.g. concurrent programming, ANTLR, StringTemplate) and the skills required in managing a large project.

Its key goals have been met, and the groundwork for future work has been laid.

# 9.

# References


[1] Y. Jiang. (2008, October) Full ANTLR grammar from OpenJDK's Compiler Grammar Project. [Online]. http://hg.openjdk.java.net/compiler-grammar/compiler-grammar/langtools/file/e37d7d5df672/src/share/classes/com/sun/tools/javac/antlr/Java.g

[2] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005.

[3] D. Patterson, "The trouble with multi-core," *IEEE Spectrum*, vol. 47, no. 7, pp. 28-32, July 2010.

[4] E. A. Lee, "The Problem with Threads," University of California, Berkerley, Berkeley, Technical Report 2006.

[5] Oracle. (2010) Java SE6 API - java.util.concurrent. [Online]. http://download.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

[6] B. Goetz et al., *Java Concurrency in Practice*. United States: Pearson Education, Inc., 2006.

[7] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, 2nd ed. NY, United States: Addison Wesley Longman, Inc., 2000.

[8] J. S. Bradbury. (2010, December) CSCI 5100G - Development of Concurrent Software Systems - University of Ontario Institute of Technology. [Online]. http://faculty.uoit.ca/bradbury/CSCI5100G.html

[9] J. Svenningsson. (2010, August) TDA381 - Concurrent Programming - University of Gothenburg. [Online]. http://www.cse.chalmers.se/edu/year/2010/course/TDA381_Concurrent_Programming/

[10] P. Welch. (2007) Concurrency Design and Practice - University of Kent. [Online]. http://www.cs.kent.ac.uk/projects/ofa/sei-cmu/

[11] B. Logan. (2008, May) G52CON Concepts of Concurrency - University of Nottingham. [Online]. http://www.cs.nott.ac.uk/~bsl/G52CON/

[12] P. Godefroid and N. Nagappan, "Concurrency at Microsoft - An exploratory survey," in *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

[13] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124-149, January 1991.

[14] M. M. Maged and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1-26, May 1998.

[15] T. Leung. (2009, July) A Survey of Concurrency Constructs: O'Reilly Open Source Convention 2009. [Online].

http://www.oscon.com/oscon2009/public/schedule/detail/8144

[16] K. Asanovic et al., "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Technical Report 2006.

[17] J. Boner. (2009) State You're Doing it Wrong: Alternative Concurrency Paradigms For the JVM - JavaOne 2009 - Slideshare. [Online]. http://www.slideshare.net/jboner/state-youre-doing-it-wrong-javaone-2009

[18] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, San Diego, California, 1993, pp. 289-300.

[19] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, New York, USA, 1995, pp. 204-213.

[20] B. Saha, A. Adl-Tabatabai, and Q. Jacobson, "Architectural Support for Software Transactional Memory," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* , Washington, DC, USA, 2006, pp. 185-196.

[21] B. C. Kuszmaul and C. E. Leiserson, "Transactions Everywhere," Massachusetts Institute of Technology Laboratory, Cambridge, Massachusetts, 2003.

[22] R. Hickey. (2010) Clojure - Refs and Transactions. [Online]. http://clojure.org/refs

[23] Project Fortress (Sun). Project Fortress - Transactions in Fortress. [Online]. http://projectfortress.sun.com/Projects/Community/wiki/TransactionsInFortress

[24] Scala STM Expert Group. (2010, December) Scala STM. [Online]. http://nbronson.github.com/scala-stm/index.html

[25] R. Hickey. (2009, October) InfoQ: Persistent Data Structures and Managed References. [Online]. http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey

[26] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1-34, March 2004.

[27] GPars. GPars Guide - Dataflow Concurrency. [Online]. http://www.gpars.org/guide/guide/7.%20Dataflow%20Concurrency.html

[28] JSR166 Expert Group. (2010) Package jsr166y - Preview versions of classes targeted for Java 7. [Online]. http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/

[29] B. Geotz. (2008) Talk: From Current to Parallel. [Online]. http://www.parleys.com/#st=5&id=25

[30] W. Pugh and J. Spacco, "MPJava: High-Performance Message Passing in Java Using Java.nio," *Lecture Notes in Computer Science*, vol. 2958, pp. 323-339, May 2004.

[31] S. Srinivasan and A. Mycroft, "Kilim: Isolation-Typed Actors for Java," in *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, Paphos, Cypress, 2008, pp. 104-128.

[32] BBC Research. (2010, November) Kamaelia. [Online]. http://www.kamaelia.org/Home.html

[33] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," *IBM Systems Journal*, vol. 41, no. 1, pp. 111-125, 2002.

[34] S. D. Stoller, "Testing Concurrent Java Programs using Randomized Scheduling," *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, pp. 142-157, December 2002.

[35] P. Joshi, M. Naik, K. Sen, and Gay D., "An effective dynamic analysis for detecting

generalised deadlocks," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2010, pp. 327-336.

[36] M. Naik, "Chord: A Static and Dynamic Program Analysis Framework for Java," Intel Labs Berkeley, Berkeley, 2010.

[37] C. Terboven, "Comparing Intel Thread Checker and Sun Thread Analyzer," in *Proceedings ParCo 2007 Conference* , 2007, pp. 668-676.

[38] SPIN. (2010, December) SPIN Model Checker - Formal Verification. [Online]. http://spinroot.com/spin/whatispin.html

[39] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A Systematic Testing Tool for Concurrent Software," Microsoft Research, Redmond, WA, Technical Report 2007.

[40] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 4, pp. 366-381, 2000.

[41] E. R., Ossietzky, C. V. Olderog, *Nets, Terms and Formulas: Three Views of Concurrent Processes and their Relationship*. Cambridge, United Kingdom: Cambridge University Press, 2005.

[42] H. Bowman and R. Gomez, *Concurrency Theory : Calculi and Automata for Modelling Untimed and Timed Concurrent Systems*. London, United Kingdom: Springer-Verlag London Limited, 2005.

[43] M. Nielsen and V. Sassone, "Lectures on Petri Nets I: Basic Models," *Petri Nets and Other Models of Concurrency*, vol. 1491, pp. 587-642, 1998.

[44] E. V. Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin, "Attribute Grammar-based Language Extensions for Java," in *Proceedings of ECOOP'07, LNCS*, 2007.

[45] OpenJDK. OpenJDK: Compiler Grammar Project. [Online]. http://openjdk.java.net/projects/compiler-grammar/

[46] A. Hristov. (2010, March) Hacking the OpenJDK Compiler. [Online]. http://www.ahristov.com/tutorial/java-compiler.html

[47] T. Parr. (2010) ANTLR Parser Generator. [Online]. http://www.antlr.org/

[48] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*.: The Pragmatic Programmers, 2007.

[49] T Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Susannah Davidson Pfalzer, Ed. USA: Pragmatic Bookshelf, 2010.

[50] A. J. Admiraal, *Automated ANTLR Tree walker Generation*.: University of Twente, 2010.

[51] BBC Research. (2010, February) Axon - the core concurrency system for Kamaelia. [Online]. http://www.kamaelia.org/Docs/Axon/Axon.html

[52] A Aho, M Lam, and R, Ullman J Sethi, *Compilers: Principles, Techniques, & Tools*, 2nd ed. USA: Pearson Addison-Wesley, 2007.

[53] T Parr. (2011, April) StringTemplate. [Online]. http://www.stringtemplate.org

[54] E Gamma, R Helm, R Johnson, and J Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Canada: Addison-Wesley, 1995.

[55] Leon Jen-Yuan Su. (2010, December) gUnit - Grammar Unit Testing - ANTLR 3. [Online]. http://www.antlr.org/wiki/display/ANTLR3/gUnit+-+Grammar+Unit+Testing

[56] C Larman, "Agile & Iterative Development: A Manager's Guide," in *Agile & Iterative Development: A Manager's Guide*. USA: Pearson Addison-Wesley, 2004, ch. 2, p. 15.

[57] S Gay, V Vasconcelos, and A Ravara, "Session Types for Inter-Process Communication," University of Glasgow, Glasgow, Technical Report 2003.

[58] K Honda, N Yoshida, and M Carbone, "Multiparty Asynchronous Session Types," in *POPL'08*, 2008, pp. 273-284.

[59] A. Tanenbaum, "Processes and Threads," in *Modern Operating Systems*. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2009, ch. 2, pp. 81-172.

# Appendix A

# Project Brief

**Student:** Stephen Tuttlebee (22632751)
**Supervisor:** Dr Pawel Sobocinski

**Project Title:**

*An Experimental Language for Concurrency:*
*Components and Synchronisation on Explicit Boundaries*

**Project Description:**

*Problem*

Concurrency has long been one of the most challenging topics in programming and in Computer Science in general. In the last few years however, its prominence has increased considerably. With Moore's Law no longer translating into performance improvement through the increase of clock speeds, the likes of Intel and AMD are instead adding more cores onto their processor chips in an effort to maintain the performance trend. Whilst adding more processing power may be easy enough, from the programmer's perspective, utilising it with the existing paradigms, languages and libraries is far less trivial. Thus, the exploration of new paradigms and enhancement of existing paradigms for concurrency has become an active area of research. As well as a search for better paradigms, there are also many efforts in the implementation of new languages and development of libraries and tools that supplement existing and emerging paradigms.

*Goals*

These streams of research all generally have the aim of simplifying concurrency for programmers. The various different languages, libraries and tools resulting from such research have had various degrees of success. This project introduces an experimental language that takes a different approach towards concurrency, and it is hoped it will be a feasible language to develop concurrent programs in. The language's core ideas were conceptualised by Dr Pawel Sobocinski. The language is based on the concepts of components and boundaries. Components can be thought of as somewhat similar to objects in typical Object-Oriented languages (e.g. Java, C++). Components declare boundaries that indicate explicitly the points through which they can communicate and synchronise with other components. The language will be a variant of Java and will initially extend a subset of it. This project will focus on developing the features and syntax of this language, expressing classic 'toy' concurrency algorithms using the language and hopefully also some more real-world examples. It is expected that the language will be constantly refined as the process of trying to express 'toy' and real examples highlights areas of improvement for the language. A compiler for the language that translates into Java and possibly even Java bytecode will also be developed.

*Scope*

This is an ambitious project and judging whether it is truly doable in the time available is difficult. If the project progresses well, the compiler mentioned above should be within the scope of this project. However there are a couple of undertakings, that although relevant and interesting, are likely to be outside the scope of this project due to time constraints. One is the development of a runtime system that dynamically optimizes execution of code written in the language. Additionally, optimizing the language and the compiler for performance will not be pursued in much depth.

# Appendix B

# Additional Language Examples

## B.1 IntProducer-IBC-IntBufferEater

In this example, IntBufferEater (IBE) simply 'eats' the values sent to it. That is, its handler is defined such that it is not possible for the IntBufferCell to block when tugging on IntBufferEater. There is no __block__; statement in the handler nor a synchronisation statement inside the handler that could cause a block to occur indirectly (which could happen via a chain of synchronisations if IntBufferEater's handler were to contain a synchronisation statement).


**Figure 48 - IntBufferEater component diagram**

```
// passive component
component IntBufferEater {
    boundary left int in?; // receives values on this boundary

    // no __run__ method

    in?[int val] {
        // eat it...or do nothing...
        eat_value(val);
    }

    private void eat_value(int value) {
        System.out.println("IntBufferEater ate value "+value);
    }
}
```

**Code Listing 48 - IntBufferEater component definition. It has one inward boundary, no __run__ method and its inward boundary's handler is defined to take and eat the value being sent to it.**

As can be seen in Code Listing 48, IntBufferEater has no __run__ method. It is an example of a passive component since 'activity' (i.e. code that is running) within the component only takes place due to synchronisations initiated by neighbouring components that invoke IntBufferEater's handlers.

## B.2 SyncCounter

This example serves to show why *the handlers of a component must be executed atomically with respect to each other*. If they were not atomic, then synchronisations initiated by other components simultaneously could lead to state inconsistencies (of the component's internal

state variables, if it has any). In this example, if a tugs occur on the out1! and out2! boundaries (being tugged by different components), then both handlers will be executed. If these tugs occur roughly simultaneously then the handlers could be run simultaneously. If no atomicity of handlers is enforced, then syncCount could get into an inconsistent state due 'unlucky' thread interleavings (since increment/decrement statements are not atomic at the Java bytecode level; they form multiple bytecode instructions).

```
// passive component
component SyncCounter {
    boundary right int out1!;
    boundary right int out2!;
    int syncCount  = 0;

    out1![int val] {
        syncCount++;
        System.out.println("syncCount increment, now equals "+syncCount);
        val = syncCount; // component wired to out1 receives value of
syncCount
    }

    out2![int val] {
        syncCount--;
        System.out.println("syncCount decremented, now equals "+syncCount);
        val = syncCount; // component wired to out2 receives value of
syncCount
    }
}
```

**Figure 49 - SyncCounter component definition**

## B.3 DiscerningIntConsumer

This is a version of IntConsumer that only consumes odd integers. Any even integers given to it are ignored. The DiscerningIntConsumer component is defined below, along with the necessary wiring/application code. The IntProducer component is simply that given previously (see Code Listing 1).

```
// active component
// NOTE: works with an ordinary IntProducer
component DiscerningIntConsumer {
    boundary left int in?;

    __run__ {
        int v;
        // non-terminating; keep accepting values
        while(true) {
            v = in?;
            // if v is not what I'm looking for (i.e. not an odd number)
            while(v % 2 == 0)
                v = in?; // tug IntProducer to get another integer
            consume_item(v);
        }
    }

    // value 'val' being pushed to us: may be even or odd
    in?[int val] {
        // consumer not-first-to-tug case: only odd values sent to us
require this component to tug back
        if (val % 2 == 1) __block__;
```

```
    }

    private void consume_item(int item) {
        System.out.println("DiscerningIntConsumer eating value "+item+", an
odd number!");
    }
}
```

**Figure 50 - DiscerningIntConsumer component definition**

This example illustrates how handlers can act like filters on the values sent (some filtering was also required in the __run__ method, to handle the case where the consumer is first to tug rather than the producer).

## B.4 LazyIntConsumer (Flag-setting)

A potential idiom for the language is 'flag-setting'. This works in the following way:

1. Component A tugs first and runs component B's handler.
2. The handler specifies to set a flag and then block. (The flag is used to indicate A's desire to synchronise with B.)
3. As B is running its run method, it continually polls the value of the flag. When the flag is set, B tugs back (by executing a synchronisation statement)

LazyIntConsumer demonstrates this for a real example:

```
/*
 * Lazy consumer that only tugs back when sender tugs it first.
 *
 * An important issue remaining is that a run method and handlers are not
 * atomic w.r.t each other. A handler (note: only one handler can execute
 * at a time) and the run method can race. Because of this racing, it
 * possible for IntConsumer  to tug first; the whole idea here is that
 * the IntConsumer is lazy, and should only ever be tugging back, never
 * tugging first.
 *
 * IN ADDITION, there are visibility problems with the senderTuggingFlag
 * variable. These can be solved by use of a volatile boolean or an
 * AtomicBoolean instead of boolean. Unfortunately, AtomicBoolean is
 * not sufficient to to provide necessary (Java) atomicity between the
 * sendertuggingFlag = true; and the __block__;.
 */
component LazyIntConsumer {
  boundary left int in?;
  boolean senderTuggingFlag; // sender is tugging, ready to give a value

  __run__ {
    while(true) {
      // flag gets read/polled in run method
      if(senderTuggingFlag) {
        int consumed = in?; // speak only when spoken to (tug only when
tugged first)
        consume_item(consumed);
        // flag also gets reset in run method after value eaten
        senderTuggingFlag = false;
      }
    }
  }

  in?[int val] {
    // flag gets set in handler
```

```
    senderTuggingFlag = true;
    // RACE CONDITION: between setting the flag and blocking, the run
    // method can see the updated value of the flag and execute the in?
    // synchronisation statement -- thus LazyIntConsumer can potentially
    // tug first
    __block__;
  }

  private void consume_item(int value) { // (E) ordinary Java method
    System.out.println("LazyIntConsumer received the value "+value);
  }
}
```

There is one concurrency issue in the above code. Although all handlers of a given component execute atomically with respect to each other, the run method and a given handler of a component do *not* execute atomically with respect to each other. Thus there can be race conditions on variables. Of course, this problem only occurs for active components (since they possess a run method).

Here there is a race condition on senderTuggingFlag. This is marked in the code above. In the handler, in between the setting of the flag and blocking, LazyIntConsumer (executing its run method) may poll senderTuggingFlag and tug the sender before the sender who was running the handler managed to block. Thus the desired behaviour of LazyIntConsumer always tugging second (tugging only when tugged) is broken when this occurs.

In fact, there is a second issue with the code above, related to memory visibility of the senderTuggingFlag.

To resolve the memory visibility issue, the senderTuggingFlag variable could be made volatile or changed to be an AtomicBoolean. However, stronger Java synchronisation is required to solve the race condition. The synchronisation mechanisms of the language are currently only designed to provide synchronisation on a wire, not within the component itself when a handler and the run method execute concurrently.

# Appendix C

# Standard Components

In addition to the synchronisation primitive standard components, Copy and Switch, there are further components that are yet to be implemented or explored in full detail. The initial work on these components is described. Initial implementations of the component definitions for these standard components can be found on the DVD-ROM (inside /JavaBTranslator/src/javab/std_comp/). The concept for each component listed here were suggestions by Sobocinski.

The typing of boundaries and components is issue that arises in this section. The type of a boundary refers to the sides, (simple) types and directions of that boundary. The type of a component refers to the types of *all* its boundaries taken together as well as the relative order of these boundaries (as defined in the component).

## C.1 Trivial Components

As seen in section 3.1.4.2, a typing constraint on starting a composition is that it has no 'outer'/'dangling' boundaries. Such dangling boundaries can artificially 'closed' by using trivial components. These trivial components contain handlers that simply either block or complete (by doing nothing). The two trivial components available are TrivialBlock and TrivialComplete:


**Figure 51 - Trivial components TrivialBlock and TrivialComplete.**

Their component definitions are given below:

```
component TrivialBlock {
    boundary left T in?;

    in?[T val] {
        __block__;
    }
}
```

**Code Listing 49 - TrivialBlock component definition. Its *in* boundary handler always blocks.**

```
component TrivialComplete {
    boundary left T in?;

    in?[T val] { }
}
```

**Code Listing 50 - TrivialComplete component definition. Its *in* boundary handler always completes without blocking.**

An example of their use is shown below:

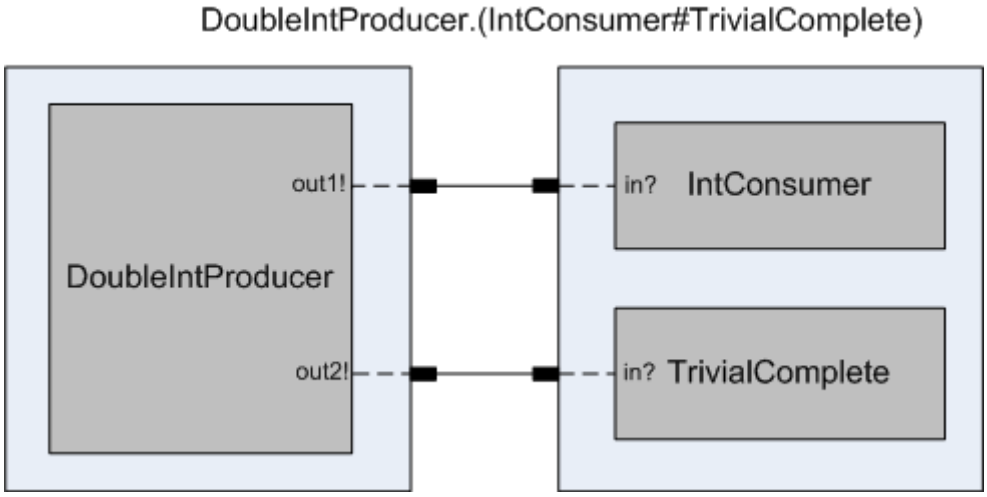


**Figure 52 - Using TrivialComplete to 'close' the other boundary of DoubleIntProducer before the composition may be started.**

To ensure typing constraints are met at 'start-time' (see section 3.1.4.2), IntConsumer is tensored with TrivialComplete before being sequentially composed with DoubleIntProducer. Doing this means ensures that all boundaries are 'closed' before the composition is started.

## C.2 Wiring Components

### C.2.1 Identity (wiring wires to wires)

An essential standard component is the Identity component:

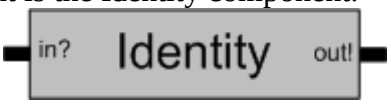


**Figure 53 - Identity component.**

This component effectively allows *wires* to be wired together. When a tug is received on its left, it simply tugs on its right. Likewise, when a tug is received on its right, it tugs on its left. It 'extends' the tug through.

The component definition for Identity follows:

```
component IdentityInwardLeft {
  boundary left T in?;
  boundary right T out!;

  // if tugged/pushed on left, tug/push on right, sending the value on
  in?[T val] {
    out![val];
  }

  // if tugged/pulled on right, tug/pull on left and then pass received
value
  out![T val] {
    val = in?;
  }
}
```

**Code Listing 51 - Identity component definition. This is actually one of the two component definitions necessary for the different typings of Identity.**

This component definition actually represents one of two possible typings for Identity. This one specifies its inward boundary on its left and outward boundary on its right. The other case is where the *outward* boundary is on its left and *inward* boundary is on its right.

An example of its use will be seen in the next section on Twist.

### C.2.2 Twist (flexible boundary order)

As described in section 3.1.4.3, there was a problem with the flexibility in wiring two components together. The order in which boundaries are defined (with respect to a certain side) 'fixes' a component's interface. Component's may only be wired to other components which have compatible 'boundary interfaces' on its left and right sides. That is, a component's left boundary interface must be compatible with the left component's right boundary interface, and similarly a component's right boundary interface must be compatible with the right component's left boundary interface.

The Twist component removes these restrictions on typing by effectively 'twisting' the wires coming in and out.

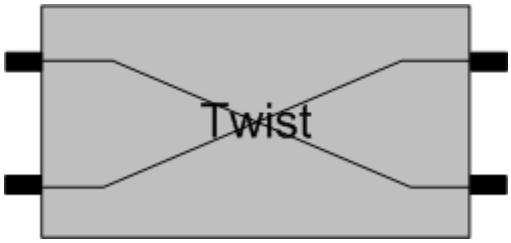


**Figure 54 - Twist component.**

In Twist, tugs on the top left boundary cause its handler to tug on the wire connected to the bottom right boundary (and vice versa). Tugs on the bottom left boundary cause its handler to tug on the wire connected to the top right boundary (and vice versa).

Of course, a single Twist component is insufficient when there is the requirement to swap boundaries by more than one 'position'. However, by using Twist in conjunction with the Identity component (along with tensor and sequential composition) several Twists can be composed together to achieve the wiring up of boundaries to any desired arbitrary order. Figure 55 gives an example:

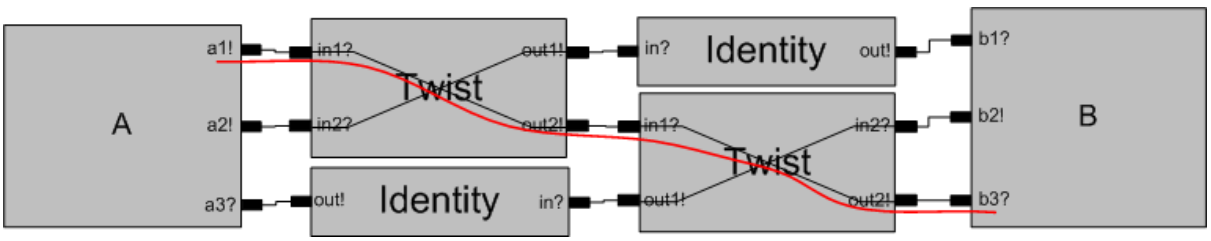


**Figure 55 - Using Twist, Identity, tensor composition and sequential composition to achieve flexible wiring between components.**

The 'path' through the middle components is shown for one particular wiring (between boundaries a1! and b3?).

In the above example, there are actually different typings of Twist and Identity used. For example the left Identity component above has out! on its left side and in? on its right, whereas the right Identity component has these reversed. A similar observation can be made for the Twist components above. In general, this is required since the directions of boundaries of components A and B are not known in advance. The appropriately typed version of the each standard component must be used (e.g. the Twist components above are two different typings of the same standard component; their functionality is equivalent but their boundary interfaces are different).

Rather than force programmers to perform all this wiring using Twist and Identity, a friendly syntax could be used to allow the programmer to just specify which boundaries should be wired with which other boundaries. The translator could then automatically insert the required Twist and Identity components.

Finally, given the current implementation, such a wiring as above would not be intolerably inefficient.

The component definition of Twist follows.

```
component Twist {
  boundary left T leftTop?;
  boundary left E leftBottom?;

  boundary right E rightTop!;
  boundary right T rightBottom!;

  // if tugged/pushed on top left, tug/push on bottom right, sending the
value on
  leftTop?[T val] {
    rightBottom[val];
  }

  // if tugged/pushed on bottom left, tug/push on top right, sending the
value on
  leftBottom?[E val] {
    rightTop![val];
  }

  // if tugged/pulled on bottom right, tug/pull on top left and then pass
received value
  rightBottom![T val] {
    val = leftTop?;
  }

  // if tugged/pulled on top right, tug/pull on bottom left and then pass
received value
  rightTop![E val] {
    val = leftBottom?;
  }
}
```

**Code Listing 52 - Twist component definition. This is actually one of four component definitions necessary for the different typings of Twist.**

This component definition actually represents one of four possible typings for Twist. This one specifies two inward boundaries on the left side and two outward boundaries on the right side.

### C.2.2.1 An Alternative to Twist

During the development of the translator, an alternative approach to achieve flexible wirings between boundaries was *originally* taken. This approach does not use Twist or any other components to achieve such wirings. Instead, the programmer specifies which boundaries should be wired to which other boundaries using an ordered list inside the composition declaration. For example:

```
      composition c = DoubleIntProducer<out2,out1>.<in1,in2>
(IntConsumer1#IntConsumer2);
```

This wires DoubleIntProducer's out2 boundary with IntConsumer1's in1 boundary, and DoubleIntProducer's out1 boundary with IntConsumer2's in2 boundary. In this example, this is effectively the same as inserting a Twist component between the two components.

With this approach, boundaries do not have their sides defined in the component definition. Instead they are determined in the wiring code. For example DoubleIntProducer's out1 and out2 boundaries could have been placed on either side, but the programmer decided to place them on the right.

Unfortunately, this approach suffers from the problem that when two components are tensored, their boundaries are combined, and this can lead to conflicting boundary names. The example above avoided this problem by having two nearly identical IntConsumer components which deliberately used different boundary names in their component definitions. One approach considered to solve this was to let the programmer rename boundaries where necessary, to avoid ambiguities:

```
      composition c = DoubleIntProducer<out2,out1>.<in as in1, in as
in2>(IntConsumer#IntConsumer);
```

This no longer requires the use of multiple IntConsumer component definitions to avoid the ambiguity. The `as` keyword is used for the renaming.

This approach has been set aside as a possible way to achieve flexible wiring, as an alternative to the current solution of using Twist.

### C.2.3 IdentityLoopback (flexible boundary sides)

One further component used for flexible wiring is the Loopback component. This component is similar to the Identity component in that it allows wires to be connected together (in fact, it is just another typing of Identity). The difference to Identity is that both boundaries appear on the same side:


**Figure 56 - IdentityLoopback component. Functionally equivalent to Identity except that both boundaries appear on same side.**

Its component definition follows:

```
component IdentityLB {
  boundary left T in?;
  boundary left T out!;

  // if tugged/pushed on top left, tug/push on bottom left, sending the
value on
  in?[T val] {
    out![val];
  }

  // if tugged/pulled on bottom left, tug/pull on top left and then pass
received value
  out![T val] {
    val = in?;
  }
}
```

**Code Listing 53 - IdentityLoopback component definition.**

The definition is the same as Identity except for a single change to the typing, that of the out! boundary, which is specified to be on the left (rather than right, as it is in Identity).

IdentityLoopback, in conjunction with Identity, can be used to swap the sides of two boundaries in a component. This is shown in the example below:

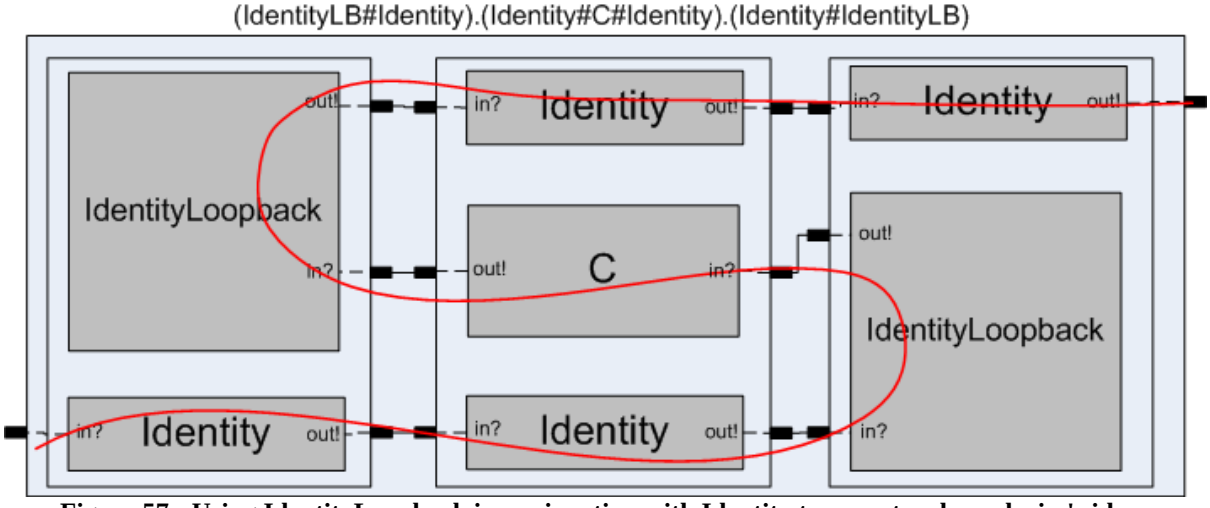


**Figure 57 - Using IdentityLoopback in conjunction with Identity to swap two boundaries' sides.**

As for the other standard components, there are multiple typings of IdentityLoopback. The two boundaries could appear be either side (both left, or both right). Secondly, the order of the boundaries could be changed.

## C.3 **Implementation Issues**

For the standard components defined here, there is the common problem of the typing of their boundaries. For each component above, it was highlighted that there are several variants for its different typings to allow it to be connected to any combination of surrounding components. These typings only considered different sides and directions. The area of the simple type of the component was ignored (e.g. int). The component definitions used generic types to indicate that there would be a need to make use of Java's generics mechanism to cater for the simple types of the boundaries the standard components are being wired to. This use of generics could be allowed at the programmer-level or only at the implementation-level, so that the translator generates code that instantiates the appropriate simple type of the standard components used using Java generics.

## C.4 **Constructing Sophisticated Components**

Combining these standard components with synchronisation primitives such as Copy and Switch, using the core operators of the language can enable the construction of components that encapsulate quite sophisticated synchronisation policies.

The following example makes use of two Copy components and an Identity component to achieve a Copy synchronisation which involves four parties rather than three:

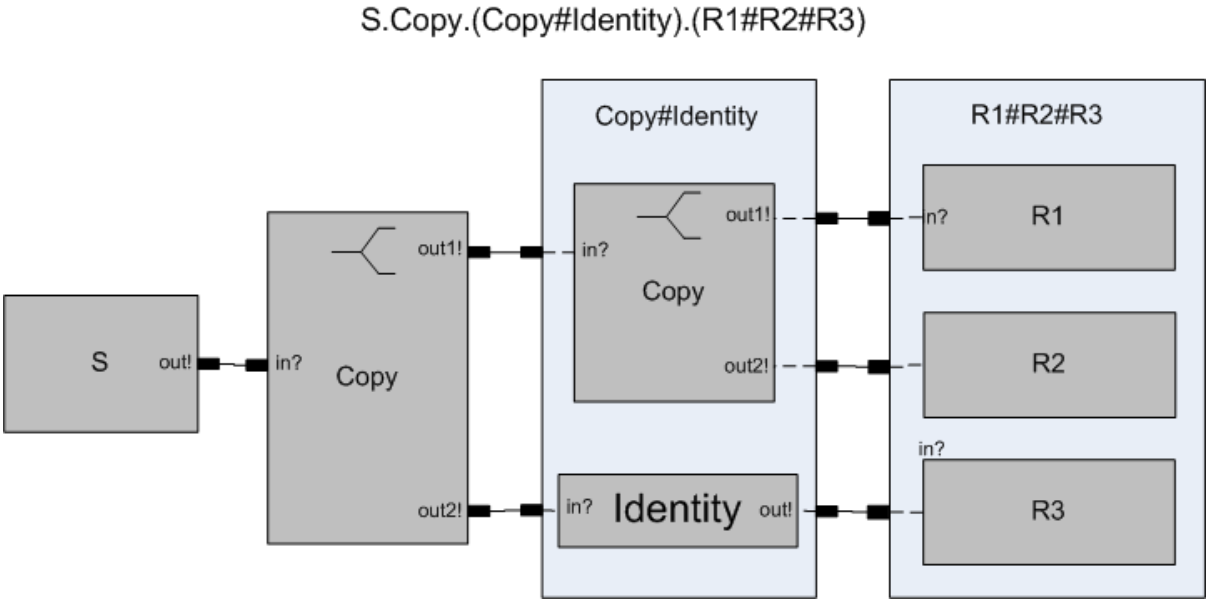


**Figure 58 - Four-party Copy synchronisation with one sender and *three* receivers, constructed using two Copy components and an Identity component.**

A similar type of topology could be used with Switch, or even a mixture of Copy and Switch. It could also potentially be extended to any number of receivers.

# Appendix D

# Translation Mechanism Classes

The source code for the Component and Boundary classes and the HandlerRunnable and Wire interfaces from section 4.1.1 are listed here. The other classes from section 4.1.1 are already listed in chapter 4.

## D.1 Component class

```
package javab.runtime;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Component superclass.
 *
 * Every component has an explicit lock associated with it that is used to
ensure execution of any handlers belonging to that component are atomic
w.r.t. each other.
 *
 * This class cannot simply implement Runnable because not all components
are active components. Passive components should definitely not be forced
be forced to implement run().
 *
 * @author Stephen J T
 */
public abstract class Component {
  private String componentName;
  private final Lock handlerLock;


  public Component(String componentName) {
    this.componentName = componentName;
    this.handlerLock = new ReentrantLock();
  }

  public Lock getLock() {
    return handlerLock;
  }

  public String getName() {
    return componentName;
  }
}
```

**Code Listing 54 - Component class used in Translation Mechanism. All generated components from the translator have this as their superclass.**

## D.2 **Boundary class**

```
package javab.runtime;

/**
 * Represents a boundary that is associated with a component. A component
may have several boundaries. Some of those boundaries are
 * 'inward', some 'outward'. The concept of 'inward' and 'outward' is not
explicitly shown here; a boundary's direction is instead indicated
 * by the way it is glued together with Wires in the main Application code
(the Wires explicitly know which boundary is the sender and which
 * is the receiver by the order in which the boundary objects are passed
into the Wire's setBoundaries(sender, receiver) method).
*/
public class Boundary<T> {
  private Component ownerComponent;
  private String name;
  private HandlerRunnable<T> handlerCode;
  private Wire<T> wireAttachedTo;

  public Boundary(String boundaryName, Component ownerComponent, Wire<T>
wireAttachedTo, HandlerRunnable<T> handler) {
    this.name = boundaryName;
    this.ownerComponent = ownerComponent;
    this.handlerCode = handler;
    this.wireAttachedTo = wireAttachedTo;
  }

  public Component getOwnerComponent() {
    return ownerComponent;
  }

  public String getBoundaryName() {
    return name;
  }

  /**
   * This method forwards the request to run the boundary's handler to the
HandlerRunnable object, stored inside this
   * Boundary object. It is forwarded to the identically named runHandler()
method inside the HandlerRunnable object
   * which contains the real handler code.
   *
   * Thus this method indirectly causes the handler associated with this
Boundary to be run by the component/thread
   * that was first to synchronise on this Boundary.
   *
   * @param value The value being passed that may or may not be used inside
the handler (depends if it is a sender or receiver).
   * @return The value (of type T) returned by the handler
   */
  public T runHandler(T value) {
    return handlerCode.runHandler(value);
  }

  /**
   * Get the wire that this boundary is attached to.
   *
   * @return the wireAttachedTo
   */
  public Wire<T> getWireAttachedTo() {
```

```
    return wireAttachedTo;
  }
}
```

**Code Listing 55 - Boundary class in used in translation mechanism.**

## D.3 HandlerRunnable interface

```
package javab.runtime;

/**
 * Use this instead of Runnable since run() in Runnable does not allow a
value to be passed as a parameter. Handlers need
 * the value sent by the sender passed to them to (optionally) use in the
code of the handler.
 *
 * Conceptually, in the semantics of the language, handlers do NOT have
parameters or return values. The reason for them
 * here however is because for the Java implementation it allows
transferring of values between components to be much simpler
 * (as well as possible!). Specifically for the sender on a Wire, it will
SUPPLY parameters and IGNORE the returned value.
 * For the receiver on a Wire, it will IGNORE parameters and USE the
returned value.
 *
 * @author Stephen J T
 *
 * @param <T>
 */
public interface HandlerRunnable<T> {
  public T runHandler(T value);
}
```

**Code Listing 56 - HandlerRunnable interface used in translation mechanism. This class is used in component classes to represent their handlers.**

## D.4 Wire interface

```
package javab.runtime;

/**
 * Run methods or handlers should be allowed to call send or receive on
 * any kind of wire. Handlers (and only handlers) must be able to call
 * blockHandler() and finishedHandler() to indicate what happened in
 * the handler.
 *
 * The only way to get round problem of having a common interface for
 * handlers and run methods to use for sending and receiving WHILST at the
 * same time being able to know which boundary invoked the receive or send
 * for CopyWire, requires that all Wires pass the sending/receiving
 * Boundary regardless of whether that specific Wire implementation needs
 * it (e.g. NormalWire doesn't need it, since it is unambiguous who called
 * the send() and receive() methods; CopyWire though does need to know who
 * called receive(), since there are two possible boundaries that called
 * it).
 *
 * @author stephen
 *
 * @param <T>
 */
public interface Wire<T> {
  // extra boundary parameter may be needed later for two sender-one
receiver version of the CopyWire, and also for SwitchWire probably
```

```
  public void send(Boundary<T> sendingBoundary, T value);
  // extra boundary parameter needed for the basic one sender-two receiver
version of CopyWire
  public T receive(Boundary<T> receivingBoundary);

  public T blockHandler(T sentValue, Boundary<T>
sendingOrReceivingBoundary, Component componentToUnlock);
  public void finishHandler(Boundary<T> sendingOrReceivingBoundary,
Component componentToUnlock);
}
```

**Code Listing 57 - Wire interface used in translation mechanism. Wire implementations such as NormalWire and CopyWire implement this interface.**

# Appendix E

# IntProducer-IBC-IntConsumer Example Translation

This section extends the discussion in section 4.1.2 with a further example.

The translation for the IntProducer-IntBufferCell-IntConsumer example is given here. The JavaB component definitions for IntProducer and IntConsumer and their Java translations are already listed and explained in section 4.1.2. Thus only the JavaB code and Java translation for IntBufferCell is listed. (This is so because the translations of components is independent of how those components may be wired together). The application/wiring code and its translation is also listed.

```
component IntBufferCell {
    // boundaries
    boundary left int in?;
    boundary right int out!;

    // internal state
    boolean empty = true;
    int value = 0;

    in?[int val] { // here 'val' is an *input parameter*
        if(empty) {
            value = val;
            empty = false;
        }
        else {
            out![value];
            value = val;
        }
    }

    out![int val] { // here 'val' is a *return parameter*
        if(!empty) {
            val = value;
            empty = true;
        }
        else {
            val = in?;
        }
    }
}
```

**Code Listing 58 - IntBufferCell component definition (IntBufferCell.javabc) (relisting of Code Listing 8).**

IntProducer-IBC-IntConsumer Example Translation

The translation of IntBufferCell.javabc follows:

```
import javab.runtime.*;

public class IntBufferCell extends Component {
  public IntBufferCell() {
    super("IntBufferCell"); // pass name of component to superclass
(Component)
  }

  // INTERNAL STATE
  private boolean empty = true;
  private int value = 0;

  // BOUNDARIES
  private Boundary<Integer> in;
  private Boundary<Integer> out;

  // HANDLERS
  public Boundary<Integer> create_boundary_in(Wire<Integer> wireAttachedTo)
{
    // the handler for this boundary

    HandlerRunnable<Integer> handler = new HandlerRunnable<Integer>() {
      public Integer runHandler(Integer val) {
        // no translator housekeeping code required before user code

        // "user code" (with JavaB parts translated) -- which could contain
a (translated) 'block;' statement
        if(empty) {
              value = val;
              empty = false;
            }
            else {
              out.getWireAttachedTo().send(out,value); // out![value]
              value = val;
            }

        // translator housekeeping code following the user code (if user
code blocks then this code is unreachable)
        in.getWireAttachedTo().finishHandler(in,IntBufferCell.this); // At
this point we know that we have finished the handler without blocking (i.e.
the sync is complete, apart from the housekeeping tasks we are about to do
now)
        return val;
        // IF OUTWARD HANDLER: it doesn't matter that we're returning back
the value the sender gave us as our dummy value for the exchanger; the
sender will ignore it anyway
        // IF INWARD  HANDLER: the handler (return) parameter val should
have been set by the programmer; if it never gets set by the programmer
then the (dummy) value that was passed in will be returned
      }
    };


    // create boundary (name, owner component, wire, handler)
    in = new Boundary<Integer>("in", this, wireAttachedTo, handler);
    return in;
  }
```

```
  public Boundary<Integer> create_boundary_out(Wire<Integer>
wireAttachedTo) {
    // the handler for this boundary

    HandlerRunnable<Integer> handler = new HandlerRunnable<Integer>() {
      public Integer runHandler(Integer val) {
        // no translator housekeeping code required before user code

        // "user code" (with JavaB parts translated) -- which could contain
a (translated) 'block;' statement
        if(!empty) {
              val = value;
              empty = true;
            }
            else {
              val = in.getWireAttachedTo().receive(in);
            }

        // translator housekeeping code following the user code (if user
code blocks then this code is unreachable)
        out.getWireAttachedTo().finishHandler(out,IntBufferCell.this); //
At this point we know that we have finished the handler without blocking
(i.e. the sync is complete, apart from the housekeeping tasks we are about
to do now)
        return val;
        // IF OUTWARD HANDLER: it doesn't matter that we're returning back
the value the sender gave us as our dummy value for the exchanger; the
sender will ignore it anyway
        // IF INWARD  HANDLER: the handler (return) parameter val should
have been set by the programmer; if it never gets set by the programmer
then the (dummy) value that was passed in will be returned
      }
    };


    // create boundary (name, owner component, wire, handler)
    out = new Boundary<Integer>("out", this, wireAttachedTo, handler);
    return out;
  }
}
```

**Code Listing 59 - Java translation of IntBufferCell component (IntBufferCell.javabc)**

As explained in section 4.1.2.1, the translation mechanisms for component definitions are generally one-to-one. For example, the internal state of the component is simply copied into the output verbatim as they are ordinary Java variables. Additionally, the two boundaries 'in' and 'out', are translated into two Boundary objects of generic parameter Integer, an autoboxing of the primitive 'int' type that the boundaries were declared as.

The translation of handlers is more complex. A handler is more than an ordinary method. It is associated with a particular Boundary object and so must be passed into its Boundary constructor. This is performed by generating a create_boundary_x() method for each boundary which the wiring code calls. The method stores the constructed Boundary inside the appropriate Boundary instance variable inside the class and also returns that Boundary object to the wiring code so that the wiring code may use it (in fact, what the wiring code does is pass the returned Boundary object into the setBoundaries() method of the NormalWire that is between two components). The Java parts of JavaB handlers do not change. Translations are only required on JavaB constructs.

IntProducer-IBC-IntConsumer Example Translation

Taking IntBufferCell's out handler as an example, a create_boundary_out() method is generated that corresponds to the Boundary that the handler is associated with. In this method, the handler code is placed inside an anonymous HandlerRunnable object which is then passed into the Boundary object constructor to create the 'out' boundary. In the handler code itself, the in? synchronisation statement in the else clause is translated into a call to receive() on the wire that the 'in' boundary is attached to. Before returning with the handler parameter (return val;), a call is made to finishHandler on the wire associated with this 'out' boundary handler to indicate that the synchronisation has completed by completing without blocking (when a handler blocks, it returns early from the handler at the point of the block and so the automatically generated finishHandler() method is not reached - see the IntProducer translated code in section 4.1.2.1).

To note here is that IntBufferCell's handlers are not defined to block as IntProducer and IntConsumer's handlers are. Therefore it does not contain any call to blockHandler() inside either of its handler 'methods'.

The JavaB wiring code and its Java translation now follow:

```
//P.IBC.C
public class Application {
    public static void main(String[] args) {
// wire IntProducer's right 'out' boundary to IBC's left 'in' boundary,
// and wire IBC's right 'out' boundary to IntConsumer's 'in' boundary
        composition c = IntProducer.IntBufferCell.IntConsumer;
        __start__ c;
    }
}
```

**Code Listing 60 - Wiring code for IntProducer-IntBufferCell-IntConsumer example (relisting of Code Listing 9).**

```
//P.IBC.C
import javab.runtime.*;

import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.CountDownLatch;

public class Application {
  public static void main(String[] args) {
    // wire up IntProducer's right 'out' boundary with IBC's left 'in'
boundary,
    // and wire up IBC's right 'out' boundary with IntConsumer's 'in'
boundary

    // create component instances contained in the composition
    IntProducer intProducer1 = new IntProducer();
    IntBufferCell intBufferCell1 = new IntBufferCell();
    IntConsumer intConsumer1 = new IntConsumer();

    // create NormalWire and CopyWire instances
    NormalWire<Integer> WIRE_intProducer1_out_TO_intBufferCell1_in = new
NormalWire<Integer>();
    NormalWire<Integer> WIRE_intBufferCell1_out_TO_intConsumer1_in = new
NormalWire<Integer>();

    // create boundary objects
```

```
    // (Boundary objects don't refer to each other, they only refer to the
Wire they are on the end of. That Wire object also has a mutual reference
to the Boundary object.)
    // use proper names for now, rather than aliases for the instance names
of Boundary objects here (the Boundary objects here remember are different
to the ones used in the translator -- they just happen to have the same
name)
    Boundary<Integer> intProducer1_out =
intProducer1.create_boundary_out(WIRE_intProducer1_out_TO_intBufferCell1_in
);
    Boundary<Integer> intBufferCell1_in =
intBufferCell1.create_boundary_in(WIRE_intProducer1_out_TO_intBufferCell1_i
n);
    Boundary<Integer> intBufferCell1_out =
intBufferCell1.create_boundary_out(WIRE_intBufferCell1_out_TO_intConsumer1_
in);
    Boundary<Integer> intConsumer1_in =
intConsumer1.create_boundary_in(WIRE_intBufferCell1_out_TO_intConsumer1_in)
;

    // now that we have created boundaries, set boundaries of the wire
object(s)

WIRE_intProducer1_out_TO_intBufferCell1_in.setBoundaries(intProducer1_out,
intBufferCell1_in);

WIRE_intBufferCell1_out_TO_intConsumer1_in.setBoundaries(intBufferCell1_out
, intConsumer1_in);

    /* Start threads of all live components (those that implement Runnable)
*/
    // use a latch 'start gate' to ensure they start at the same time --
see JCIP chapter 5)
    final CountDownLatch startGate = new CountDownLatch(1);

    // add all Runnables to a set to be iterated over
    Set<Runnable> runnables = new HashSet<Runnable>();
    runnables.add(intProducer1);
    runnables.add(intConsumer1);

    // set of latch-altered Runnables that have been turned into Threads
    Set<Thread> threads = new HashSet<Thread>();

    // iterate over them and wrap their run methods to include
startGate.await() at the beginning
    for(final Runnable r : runnables) {
      Thread t = new Thread() {
        public void run() {
          try {
            startGate.await();
            r.run();
          }
          catch(InterruptedException e) { e.printStackTrace(); }
        }
      };
      threads.add(t);
      t.start(); // also start the thread (it will await at latch)
    }

    // GO! (release all the threads)
```

```
    startGate.countDown();
  }
}
```

**Code Listing 61 - Java Translation of wiring code above (Application.javab).**

The wiring code here is very similar to that in section 4.1.2.2. Here an IntBufferCell object is also instantiated. Additionally, two more Boundary objects are created which correspond to the two boundaries of the IntBufferCell that has been introduced into the composition. Another point to note is that there is an extra NormalWire object and the NormalWire objects end-point boundaries are set to different Boundary objects to that seen in section 4.1.2.2, since there is an extra component and a different wiring between them.

# Appendix F

# Wire Implementations

## F.1 NormalWire Implementation

```
package javab.runtime;

import java.util.concurrent.Exchanger;
import java.util.concurrent.atomic.AtomicInteger;

public class NormalWire<T> implements Wire<T> {
  private final Object wireLock;
  private Boundary<T> sender;
  private Boundary<T> receiver;
  private final Exchanger<T> valueExchanger;
  @GuardedBy("wireLock") private boolean syncIncomplete;
  private volatile boolean handlerBlocked;

  private final Object sendMethodLock;
  private final Object receiveMethodLock;

  private AtomicInteger numThreadsOnWire; // 0 <= x <= 2

  public NormalWire() {
    this.wireLock = new Object();

    this.sender = null;
    this.receiver = null;

    this.valueExchanger = new Exchanger<T>();
    this.syncIncomplete = false;
    this.handlerBlocked = false;

    this.sendMethodLock = new Object();
    this.receiveMethodLock = new Object();

    this.numThreadsOnWire = new AtomicInteger(0);
  }

  public synchronized void setBoundaries(Boundary<T> sender, Boundary<T> receiver)
{
    this.sender = sender;
    this.receiver = receiver;
  }

  public void send(Boundary<T> sendingBoundary, T value) {
    synchronized(sendMethodLock) { // acquire "send lock" so that no two threads
can do a send() at the same time

      // break the symmetry
      boolean runTheHandler;
      synchronized(wireLock) {
        runTheHandler = !syncIncomplete;
        syncIncomplete = true;
```

```
      numThreadsOnWire.incrementAndGet();
    }


    // first to tug
    if(runTheHandler) {
      // possibility of not being able to acquire component's lock
      boolean done = false;
      while(!done) {
        // if we succeed in grabbing the lock
        if(receiver.getOwnerComponent().getLock().tryLock()) {
          receiver.runHandler(value);

          done = true;
        }
        // if we fail to grab the lock
        else {
          if(numThreadsOnWire.get() == 1) {
            Thread.yield();
          }
          if(numThreadsOnWire.get() == 2) {
            // pretend we were running a handler and blocked
            synchronized(wireLock) {
              this.handlerBlocked = true;
              wireLock.notifyAll();
            }

            // proceed to the exchange
            try { valueExchanger.exchange(value); }
            catch (InterruptedException e) { e.printStackTrace(); }

            // decrement numThreadsOnWire now that exchange/sync is done
            numThreadsOnWire.decrementAndGet();

            done = true;
          }
        }
      }
    }
    // second to tug
    else {
      boolean startNewSync;
      synchronized(wireLock) {
        // while handler has neither finished nor blocked, wait
        while(syncIncomplete && !handlerBlocked) {
          // wait until notified
          try { wireLock.wait(); }
          catch (InterruptedException e) { e.printStackTrace(); }
        }

        // if the handler blocked, don't start new sync
        if(handlerBlocked)
          startNewSync = false;
        // else handler must have completed
        else
          startNewSync = true;

        // reset flags
        syncIncomplete = false;
        handlerBlocked = false;
      }

      // if the sync completed, then start a new sync ('re-tug')
      if(startNewSync) {
        // decrement BEFORE doing the recursive call
        numThreadsOnWire.decrementAndGet();
        send(sendingBoundary,value);
```

```
        }
        // otherwise, the handler must have blocked, so meet at exchanger
        else {
          try { valueExchanger.exchange(value); }
          catch (InterruptedException e) { e.printStackTrace(); }

          // decrement numThreadsOnWire now that exchange/sync is done
          numThreadsOnWire.decrementAndGet();
        }
      }
    }
  }

  public T receive(Boundary<T> receivingBoundary) {
    T valueReceived = null;
    synchronized(receiveMethodLock) {

      // break the symmetry
      boolean runTheHandler;
      synchronized(wireLock) {
        runTheHandler = !syncIncomplete;
        syncIncomplete = true;

        numThreadsOnWire.incrementAndGet();
      }

      // first to tug
      if(runTheHandler) {
        // possibility of not being able to acquire component's lock
        boolean done = false;
        while(!done) {
          // if we succeed in grabbing the lock
          if(sender.getOwnerComponent().getLock().tryLock()) {
            valueReceived = sender.runHandler(null);
            done = true;
          }
          // if we fail to grab the lock
          else {
            if(numThreadsOnWire.get() == 1) {
              Thread.yield();
            }
            if(numThreadsOnWire.get() == 2) {
              // pretend we were running a handler and that we blocked
              synchronized(wireLock) {
                this.handlerBlocked = true;
                wireLock.notifyAll();
              }

              // proceed to the exchange
              try { valueReceived = valueExchanger.exchange(null); }
              catch (InterruptedException e) { e.printStackTrace(); }

              // decrement numThreadsOnWire now that exchange/sync is done
              numThreadsOnWire.decrementAndGet();

              done = true;
            }
          }
        }
      }
      // second to tug
      else {
        boolean startNewSync;
        synchronized(wireLock) {
          // while handler has neither finished nor blocked, wait
          while(syncIncomplete && !handlerBlocked) {
            // wait until notified
            try { wireLock.wait(); }
```

```
            catch (InterruptedException e) { e.printStackTrace(); }
          }

          // if the handler blocked then don't start new sync
          if(handlerBlocked)
            startNewSync = false;
          // else the handler must have completed
          else
            startNewSync = true;

          // reset flags
          syncIncomplete = false;
          handlerBlocked = false;
        }

        // if sync completed, then need to start a new tug
        if(startNewSync) {
          // decrement BEFORE doing the recursive call
          numThreadsOnWire.decrementAndGet();
          valueReceived = receive(receivingBoundary);
        }
        // otherwise, the handler must have blocked
        else {
          try { valueReceived = valueExchanger.exchange(null); }
          catch (InterruptedException e) { e.printStackTrace(); }

          // decrement numThreadsOnWire now that exchange/sync is done
          numThreadsOnWire.decrementAndGet();
        }
      }
    }
    return valueReceived;
  }

  /**
   * Called by a handler when it encounters a __block__; statement.
   * It will cause the component running the handler to block at
   * the Exchanger until the other componen tugs back via a call
   * to valueExchanger.exhange().
   */
  public T blockHandler(T sentValue, Boundary<T> sendingOrReceivingBoundary,
Component componentToUnlock) {
    synchronized(wireLock) {
      numThreadsOnWire.decrementAndGet();
      componentToUnlock.getLock().unlock();
      this.handlerBlocked = true;
      wireLock.notifyAll();
    }

    T valueReceived = null;
    try { valueReceived = valueExchanger.exchange(sentValue); }
    catch (InterruptedException e) { e.printStackTrace(); }
    return valueReceived;
  }

  /**
   * Called by a handler to notify us when it has finished
   * without blocking.
   */
  public void finishHandler(Boundary<T> sendingOrReceivingBoundary, Component
componentToUnlock) {
    synchronized(wireLock) {
      numThreadsOnWire.decrementAndGet();
      componentToUnlock.getLock().unlock();
      this.syncIncomplete = false;
      wireLock.notifyAll();
    }
  }
```

```
}
```

**Code Listing 62 - NormalWire Java Implementation**

## F.2 CopyWire Implementation

```
package javab.runtime;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class CopyWire<T> implements Wire<T> {
  private final Object wireLock;

  private Boundary<T> sender;
  private Boundary<T> receiver1;
  private Boundary<T> receiver2;
  private final ExecutorService exec; // executor for thread pool of threads that
run the handlers
  private final CyclicBarrier barrier;

  private volatile T valueToTransfer;
  @GuardedBy("wireLock") private boolean syncIncomplete; // state of entire
synchronisation
  private final Object syncIncompleteCondition; // condition variable on state of
syncIncomplete

  // partial synchronisations between subgroups of the 3 parties (i.e. between sets
of two parties)
  // state of "partial" synchronizations
  @GuardedBy("wireLock") private boolean[] partialSyncHandlerFinished;
  @GuardedBy("wireLock") private boolean[] partialSyncHandlerBlocked;
  private Semaphore[] handlerBlockedConditionVar;

  // mappings of boundaries to indices into the arrays above
  private Map<Boundary<T>,Integer> boundaryToArrayIndex;

  private boolean sendersHandlerRun; // has sender's handler run? (n/a when sender
first to tug)
  private Object sendersHandlerRunCondition; // condition variable on above
variable

  public CopyWire() {
    this.wireLock = new Object();
    this.sender = null;
    this.receiver1 = null;
    this.receiver2 = null;
    this.valueToTransfer = null;
    this.exec = Executors.newFixedThreadPool(2);

    this.barrier = new CyclicBarrier(3, new Runnable() {
      public void run() {
        // mark the synchronisation as now being complete
        syncIncomplete = false;
        synchronized(syncIncompleteCondition) {
          // may be up to 2 threads waiting on this condition variable
          // (if both second and third tuggers were 'late' for current
synchronization)
          syncIncompleteCondition.notifyAll();
        }
      }
    });
```

```
    this.syncIncomplete = false;
    this.syncIncompleteCondition = new Object();

    this.partialSyncHandlerFinished = new boolean[3];
    this.partialSyncHandlerBlocked = new boolean[3];
    this.handlerBlockedConditionVar = new Semaphore[3];

    // initialise values
    for(int i = 0; i < 3; i++) {
      this.partialSyncHandlerFinished[i] = false;
      this.partialSyncHandlerBlocked[i] = false;
      this.handlerBlockedConditionVar[i] = new Semaphore(0);
    }

    this.boundaryToArrayIndex = new HashMap<Boundary<T>,Integer>();

    this.sendersHandlerRun = false;
    this.sendersHandlerRunCondition = new Object();
  }

  public synchronized void setBoundaries(Boundary<T> sender, Boundary<T> receiver1,
Boundary<T> receiver2) {
    this.sender = sender;
    this.receiver1 = receiver1;
    this.receiver2 = receiver2;

    // initialise mappings here because sender, receiver1 and receiver2 not
available until setBoundaries is called
    this.boundaryToArrayIndex.put(this.sender, 0);
    this.boundaryToArrayIndex.put(this.receiver1, 1);
    this.boundaryToArrayIndex.put(this.receiver2, 2);
  }


  // (sendingBoundary parameter only required for the other version CopyWire with
two senders and one receiver)
  public void send(Boundary<T> sendingBoundary, final T value) {
    valueToTransfer = value;

    // do..while used here as a better means of starting a new sync.
    boolean startNewSync;
    do {
      // initialise/reset startNewSync
      startNewSync = false;

      // break the symmetry
      boolean firstToTug;
      synchronized(wireLock) {
        firstToTug = !syncIncomplete;
        syncIncomplete = true;
      }

      // if the SENDER WAS FIRST to tug {Development Note: 'tryLock' stuff removed
for simplicity; thus deadlock may be possible}
      if(firstToTug) {
        // run both receivers' handlers in separate threads

        // start thread pool going!
        exec.execute(new Runnable() {
          public void run() {
            receiver1.getOwnerComponent().getLock().lock();
            receiver1.runHandler(valueToTransfer); // returns dummy value that we
ignore
            // at this point, receiver1's handler has either finished without
blocking or it has blocked-then-unblocked-by-receiver1
            // unlock occurs in blockHandler() or finishedHandler() methods
```

```
            // if handler finished without blocking, then wait at the barrier (in
this thread)
            boolean handlerFinishedWithoutBlocking;
            int index = boundaryToArrayIndex.get(receiver1);
            synchronized(wireLock) {
              handlerFinishedWithoutBlocking = partialSyncHandlerFinished[index];
            }

            if(handlerFinishedWithoutBlocking) {
              try { barrier.await(); }
              catch (InterruptedException e) { e.printStackTrace(); }
              catch (BrokenBarrierException e) { e.printStackTrace(); }
            }
          }
        });
        exec.execute(new Runnable() {
          public void run() {
            receiver2.getOwnerComponent().getLock().lock();
            receiver2.runHandler(valueToTransfer); // returns dummy value that we
ignore
            // at this point, receiver2's handler has either finished without
blocking or it has blocked-then-unblocked-by-receiver2
            // unlock occurs in blockHandler() or finishedHandler() methods

            // if handler finished without blocking, then wait at the barrier (in
this thread)
            boolean handlerFinishedWithoutBlocking;
            int index = boundaryToArrayIndex.get(receiver2);
            synchronized(wireLock) {
              handlerFinishedWithoutBlocking = partialSyncHandlerFinished[index];
            }

            if(handlerFinishedWithoutBlocking) {
              try { barrier.await(); }
              catch (InterruptedException e) { e.printStackTrace(); }
              catch (BrokenBarrierException e) { e.printStackTrace(); }
            }
          }
        });

        // wait at barrier until release is caused by everyone arriving
        try { barrier.await(); }
        catch (InterruptedException e) { e.printStackTrace(); }
        catch (BrokenBarrierException e) { e.printStackTrace(); }
        // when barrier trips, imagine barrier action happening here...
      }
      // if SENDER WAS SECOND OR THIRD to tug
      else {
        // (NOTE: no running of handlers takes place)

        int index = boundaryToArrayIndex.get(sendingBoundary);

        // wait and see whether the handlers run by the first guy finished or
blocked
        boolean handlerBlocked,handlerFinished;
        synchronized(wireLock) {
          // while handler has neither finished nor blocked, wait
          while(!partialSyncHandlerFinished[index] &&
!partialSyncHandlerBlocked[index]) {
            // wait until notified
            try { wireLock.wait(); }
            catch (InterruptedException e) { e.printStackTrace(); }
          }

          handlerBlocked = partialSyncHandlerBlocked[index];
          handlerFinished = partialSyncHandlerFinished[index];
        }
```

```
        if(handlerBlocked) {
          // unblock them
          handlerBlockedConditionVar[index].release();
          // wait at barrier (we DON'T wait at the barrier in the case where they
finished the handler)
          try { barrier.await(); }
          catch (InterruptedException e) { e.printStackTrace(); }
          catch (BrokenBarrierException e) { e.printStackTrace(); }
        }

        if(handlerFinished) {
          startNewSync = true;

          synchronized(syncIncompleteCondition) {
            // wait until entire synchronization is complete; when it is complete
then at that point start a new sync
            while(syncIncomplete) {
              try {
                syncIncompleteCondition.wait();
              }
              catch (InterruptedException e) { e.printStackTrace(); }
            }
          }
          // the above will only be released from wait until the barrier action has
been executed
        }

        synchronized(wireLock) {
          // reset flags (only one of these will truly be necessary)
          partialSyncHandlerFinished[index] = false;
          partialSyncHandlerBlocked[index] = false;
        }
      }
    } while(startNewSync);

    // returns nothing
  }

  // the two receivers 'share' (i.e. both call) this method
  public T receive(Boundary<T> receivingBoundary) {
    // do..while used here as a better means of starting a new sync.
    boolean startNewSync;
    do {
      // initialise/reset startNewSync
      startNewSync = false;

      // break the symmetry
      boolean firstToTug;
      synchronized(wireLock) {
        firstToTug = !syncIncomplete;
        syncIncomplete = true;
      }

      // if this receiver WAS FIRST to tug {Development Note: 'tryLock' stuff
removed for simplicity; thus deadlock may be possible}
      if(firstToTug) {
        exec.execute(new Runnable() {
          public void run() {
            sender.getOwnerComponent().getLock().lock();
            valueToTransfer = sender.runHandler(null);
            // unlock occurs in blockHandler() or finishedHandler() methods

            // notify the other spawned thread that will run the other receiver's
handler that the sender's handler has been run
            synchronized(sendersHandlerRunCondition) {
              sendersHandlerRun = true;
              sendersHandlerRunCondition.notifyAll();
            }
```

```
            // if handler finished without blocking, then wait at the barrier (in
this thread)
            boolean handlerFinishedWithoutBlocking;
            int index = boundaryToArrayIndex.get(sender);
            synchronized(wireLock) {
              handlerFinishedWithoutBlocking = partialSyncHandlerFinished[index];
            }

            if(handlerFinishedWithoutBlocking) {
              try { barrier.await(); }
              catch (InterruptedException e) { e.printStackTrace(); }
              catch (BrokenBarrierException e) { e.printStackTrace(); }
            }
          }
        });

        // determine who the other receiver is who's handler will need to be run
        final Boundary<T> otherReceiver;
        if(receivingBoundary == receiver1) { otherReceiver = receiver2; }
        else if(receivingBoundary == receiver2) { otherReceiver = receiver1; }
        else { throw new AssertionError("The boundary passed to the receive()
method was neither of the receiving boundaries specified originally."); }

        exec.execute(new Runnable() {
          public void run() {
            // wait until sender's handler has been run and its return value set to
valueToTransfer
            synchronized(sendersHandlerRunCondition) {
              while(!sendersHandlerRun) {
                try {
                  sendersHandlerRunCondition.wait();
                }
                catch (InterruptedException e) { e.printStackTrace(); }
              }
              // reset flag for the next sync
              sendersHandlerRun = false;
            }

            otherReceiver.getOwnerComponent().getLock().lock();
            otherReceiver.runHandler(valueToTransfer);
            // unlock occurs in blockHandler() or finishedHandler() methods

            // if handler finished without blocking, then wait at the barrier (in
this thread)
            boolean handlerFinishedWithoutBlocking;
            int index = boundaryToArrayIndex.get(otherReceiver);
            synchronized(wireLock) {
              handlerFinishedWithoutBlocking = partialSyncHandlerFinished[index];
            }

            if(handlerFinishedWithoutBlocking) {
              try { barrier.await(); }
              catch (InterruptedException e) { e.printStackTrace(); }
              catch (BrokenBarrierException e) { e.printStackTrace(); }
            }
          }
        });

        // wait at barrier until release is caused by everyone arriving
        try { barrier.await(); }
        catch (InterruptedException e) { e.printStackTrace(); }
        catch (BrokenBarrierException e) { e.printStackTrace(); }
        // when barrier trips, imagine barrier action happening here...
      }
      // if this receiver WAS SECOND OR THIRD to tug
      else {
        // NOTE: no running of handlers takes place
```

```
        int index = boundaryToArrayIndex.get(receivingBoundary);

        // wait and see whether the handlers run by the first guy finished or
blocked (the first guy was either the sender or the other receiver)
        boolean handlerBlocked,handlerFinished;
        synchronized(wireLock) {
          // while handler has neither finished nor blocked, wait
          while(!partialSyncHandlerFinished[index] &&
!partialSyncHandlerBlocked[index]) {
            // wait until notified
            try { wireLock.wait(); }
            catch (InterruptedException e) { e.printStackTrace(); }
          }

          handlerBlocked = partialSyncHandlerBlocked[index];
          handlerFinished = partialSyncHandlerFinished[index];
        }

        if(handlerBlocked) {
          // unblock them
          handlerBlockedConditionVar[index].release();
          // wait at barrier (we DON'T wait at the barrier in the case where they
finished the handler)
          try { barrier.await(); }
          catch (InterruptedException e) { e.printStackTrace(); }
          catch (BrokenBarrierException e) { e.printStackTrace(); }
        }

        if(handlerFinished) {
          startNewSync = true;

          synchronized(syncIncompleteCondition) {
            // wait until entire synchronization is complete; when it is complete
then at that point start a new sync
            while(syncIncomplete) {
              try {
                syncIncompleteCondition.wait();
              }
              catch (InterruptedException e) { e.printStackTrace(); }
            }
          }
          // the above will only be released from wait until the barrier action has
been executed
        }

        synchronized(wireLock) {
          // reset flags (only one of these will truly be necessary)
          partialSyncHandlerFinished[index] = false;
          partialSyncHandlerBlocked[index] = false;
        }
      }
    } while(startNewSync);

    return valueToTransfer;
  }

  /**
   * Called by handlers that complete the synchronization by blocking (although the
synchronization won't be complete until unblocked).
   *
   * This method may be called by a handler that corresponds to a sending or
receiving boundary -- hence there is both a parameter and a return value for the
   * sent and received values. A handler that calls this method will only make use
of the parameter or the return value, but not both.
   *
   * This method and the finishedHandler() method get called twice between them in
a single synchronization -- once by the second tugger and once by the third tugger.
```

```
    *
    * This could be called simultaneously by the 2 threads that are running the
handlers e.g. if the sender tugged first, then there could be two threads
    * running receiver1 and receiver2's handlers.
    *
    * @param sentValue Value to be sent (only meaningful for senders)
    * @param boundary The sending or receiving boundary that is calling this
blockHandler method. Only used in CopyWire (not used/needed in NormalWire).
    * @param componentToUnlock The component that owns the handler (who's associated
lock needs to be released before the actual blocking occurs)
    * @return Value to be received (only meaningful for receivers)
    */
  public T blockHandler(T sentValue, Boundary<T> boundary, Component
componentToUnlock) {
    // appropriate index into arrays
    int index = boundaryToArrayIndex.get(boundary);

    synchronized(wireLock) {
      componentToUnlock.getLock().unlock();
      // indicate that sync will complete by blocking-then-unblocking (rather than
finishing the handler)
      this.partialSyncHandlerBlocked[index] = true;
      wireLock.notifyAll();
    }

    // BLOCK (or don't block if someone's already ready for us; because this is a
semaphore)
    handlerBlockedConditionVar[index].acquireUninterruptibly();

    return valueToTransfer;
  }


  /**
    * Called by a handlers that complete the synchronisation by finishing the
handler without blocking.
    *
    * This method and the blockHandler() method get called twice between them in a
single synchronisation -- once by the second tugger and once by the third tugger.
    *
    * @param boundary The sending or receiving boundary that is calling this
finishedHandler method. Only used in CopyWire (not used/needed in NormalWire).
    * @param componentToUnlock The component that owns the handler (who's associated
lock needs to be released)
    */
  public void finishHandler(Boundary<T> boundary, Component componentToUnlock) {
    // appropriate index into arrays
    int index = boundaryToArrayIndex.get(boundary);

    synchronized(wireLock) {
      componentToUnlock.getLock().unlock();
      this.partialSyncHandlerFinished[index] = true;
      wireLock.notifyAll();
    }
  }

  /**
    * Necessary method for the main application thread to call when all the normal
program threads have terminated and the only remaining nondaemon
    * threads are those in the Executor thread pool of this CopyWire. Thus a
shutdown method needs to be exposed so that the program can
    * ask for the executor / thread pool to be shutdown. This is so that the JVM may
exit -- the JVM does not exit until all nondaemon threads have
    * terminated.
    */
  public void shutdownExecutor() {
    // normal approach of awaitTermination() and shutdown() not working
    // temporary solution
```

```
    exec.shutdownNow();
  }
}
```

**Code Listing 63 - CopyWire Java Implementation**

## F.3 **Flow charts of NormalWire**

The original NormalWire implementation suffered from deadlock. The fundamental cause of the deadlock is due to the acquisition of a component's locks, which must be acquired before running any handler of that component. Component locks are acquired to ensure atomicity of handler execution with respect to other handlers in the component.

The following subsections show flow charts representing the NormalWire logic for both the deadlock-prone and deadlock-free versions of NormalWire. An example is also provided.

### F.3.1 Flow charts for Deadlock-Prone NormalWire

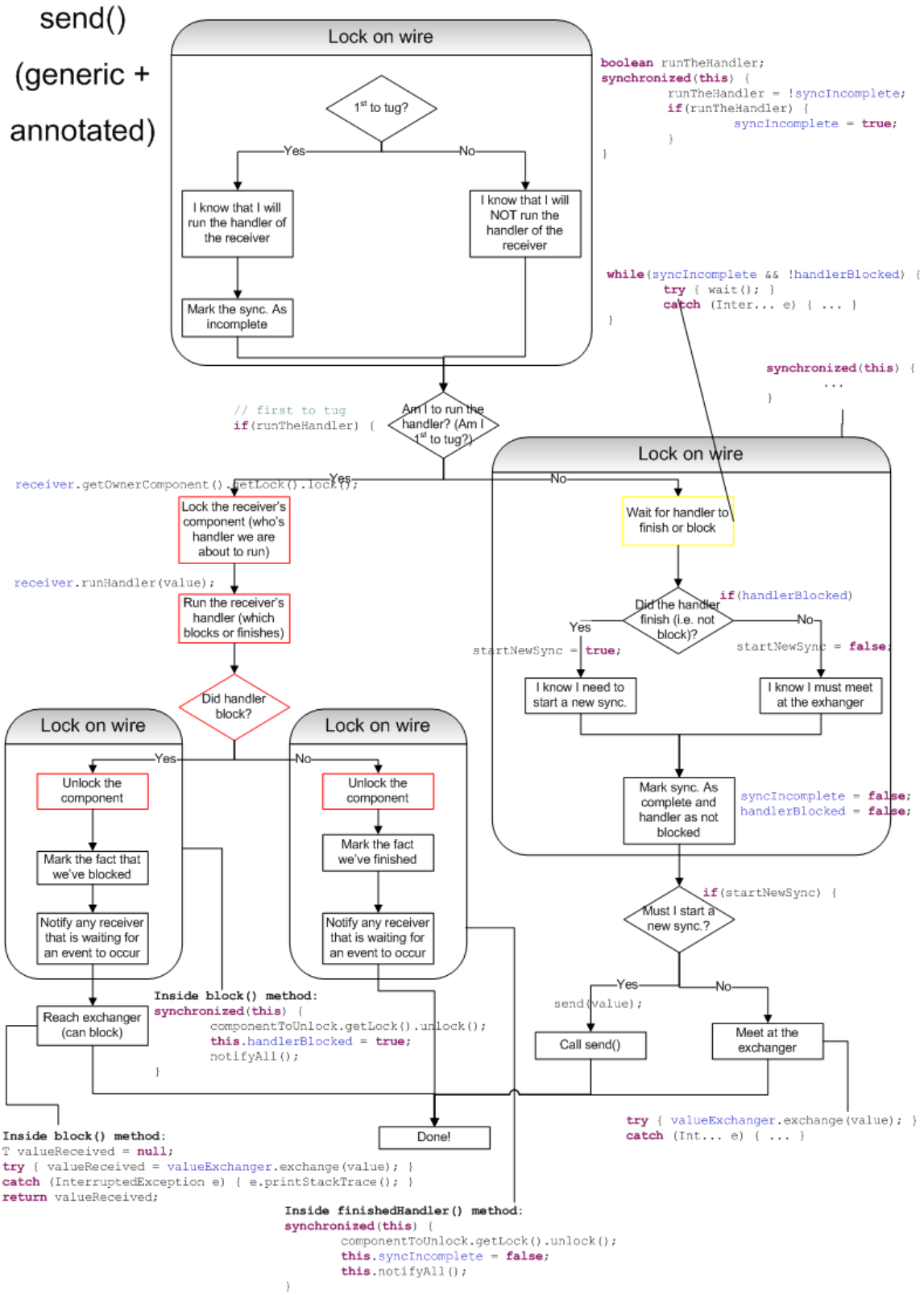


**Figure 59 - Flow chart showing logic of send() method of deadlock-prone NormalWire, annotated with corresponding code fragments**

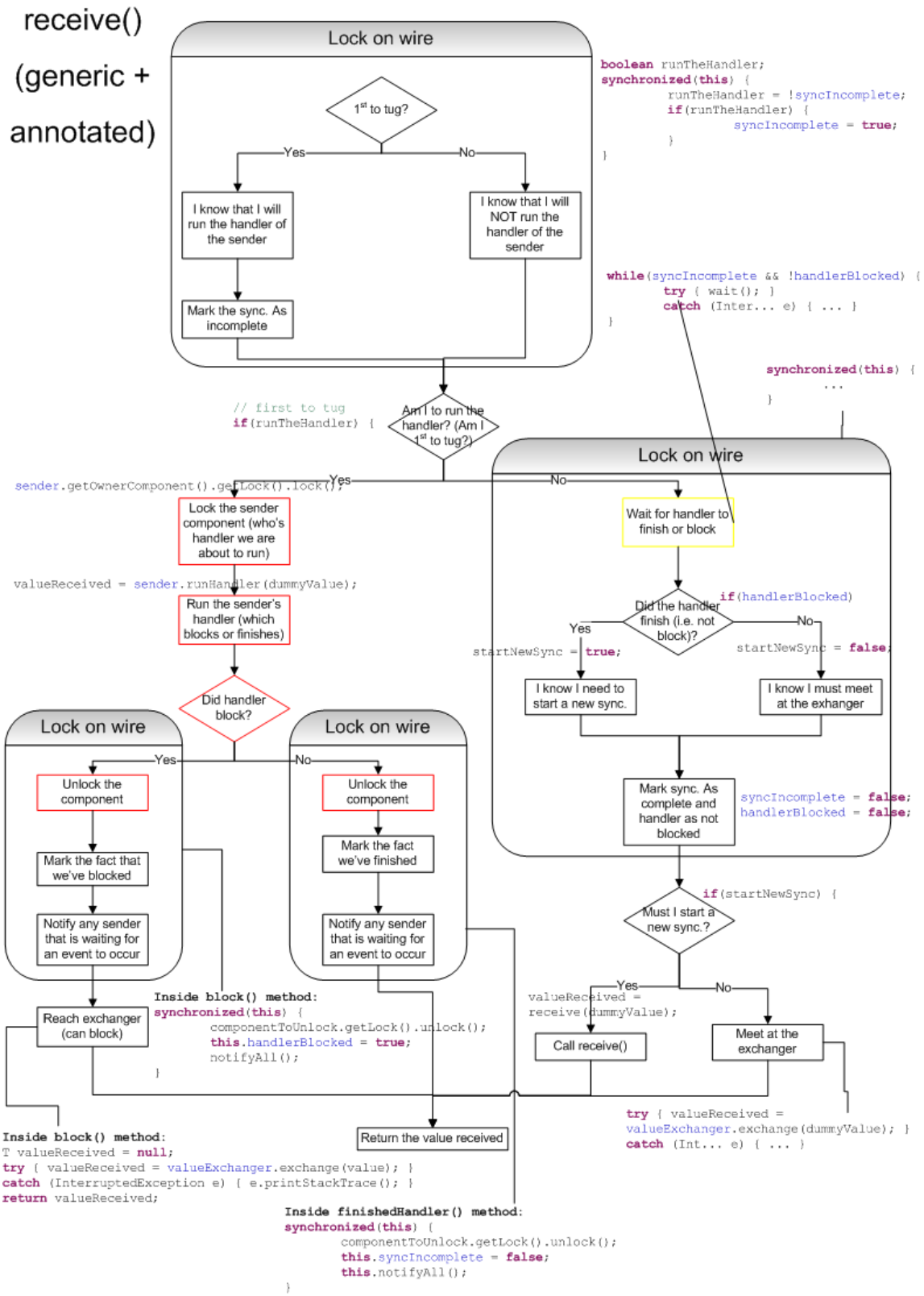


**Figure 60 - Flow chart showing logic of receive() method of deadlock-prone NormalWire, annotated with corresponding code fragments.**

### F.3.2 Example Deadlock

A specific scenario demonstrating how deadlock can occur in the deadlock-prone version of NormalWire is now given using the IntProducer-IntBufferCell-IntConsumer example:

[PRODUCER THREAD] producer about to send()
[CONSUMER THREAD] consumer about to receive()
[PRODUCER THREAD] IntProducer1 (the sender) was first to tug and so is going to run the handler (but it must first succeed in acquiring the component lock of intBufferCell1 (the receiver))
[CONSUMER THREAD] IntConsumer1 (the receiver) was first to tug and so is going to run the handler (but it must first succeed in acquiring the component lock of intBufferCell1 (the sender))
[CONSUMER THREAD] Someone is running intBufferCell1's handler when trying to pull a value from it
[CONSUMER THREAD] IBC intBufferCell1 is empty, so it will first receive() on its neighbour in order to get a value which it will then return to the pulling component. The IBC remains empty throughout.
[CONSUMER THREAD] intBufferCell1 (the receiver) is second to tug and so is NOT going to run the handler but is going to wait to see whether the sender finishes the handler (sync is complete) or blocks in the handler (unblocking and transfer of value required and then sync is complete).
***DEADLOCK!***

Both IntProducer and IntConsumer are first to tug on their respective wires, with IntBufferCell in the middle. IntConsumer beats IntProducer1 to acquiring IntBufferCell's component lock and so may run IntBufferCell's (out!) handler. IntProducer1 is essentially left blocked on the P-IBC wire for IntBufferCell's component lock to be released so that InProducer can run IntBufferCell's (in?) handler. Meanwhile, when IntConsumer runs the out! handler, a synchronisation statement in the handler is encountered which causes a tug on the P-IBC wire. Since IntProducer is already first on that wire, the consumer thread will wait on the wire, expecting that IntProducer is busily running a handler in IntBufferCell. In reality, IntProducer is waiting for the IntBufferCell component lock so that it can run the handler. Thus the producer thread is waiting on a component lock which is held by the consumer thread. The consumer thread is waiting on the producer thread to finish running the IntBufferCell's in? handler, which cannot happen until the component lock is released. DEADLOCK.

This scenario is also illustrated in the following flow charts. The flow charts should be taken together as the same execution of the program. The order of steps taken are numbered sequentially, with concurrent steps indicated with additional letters to distinguish them (e.g. 1a, 1b). Execution begins in Figure 61 and Figure 62.

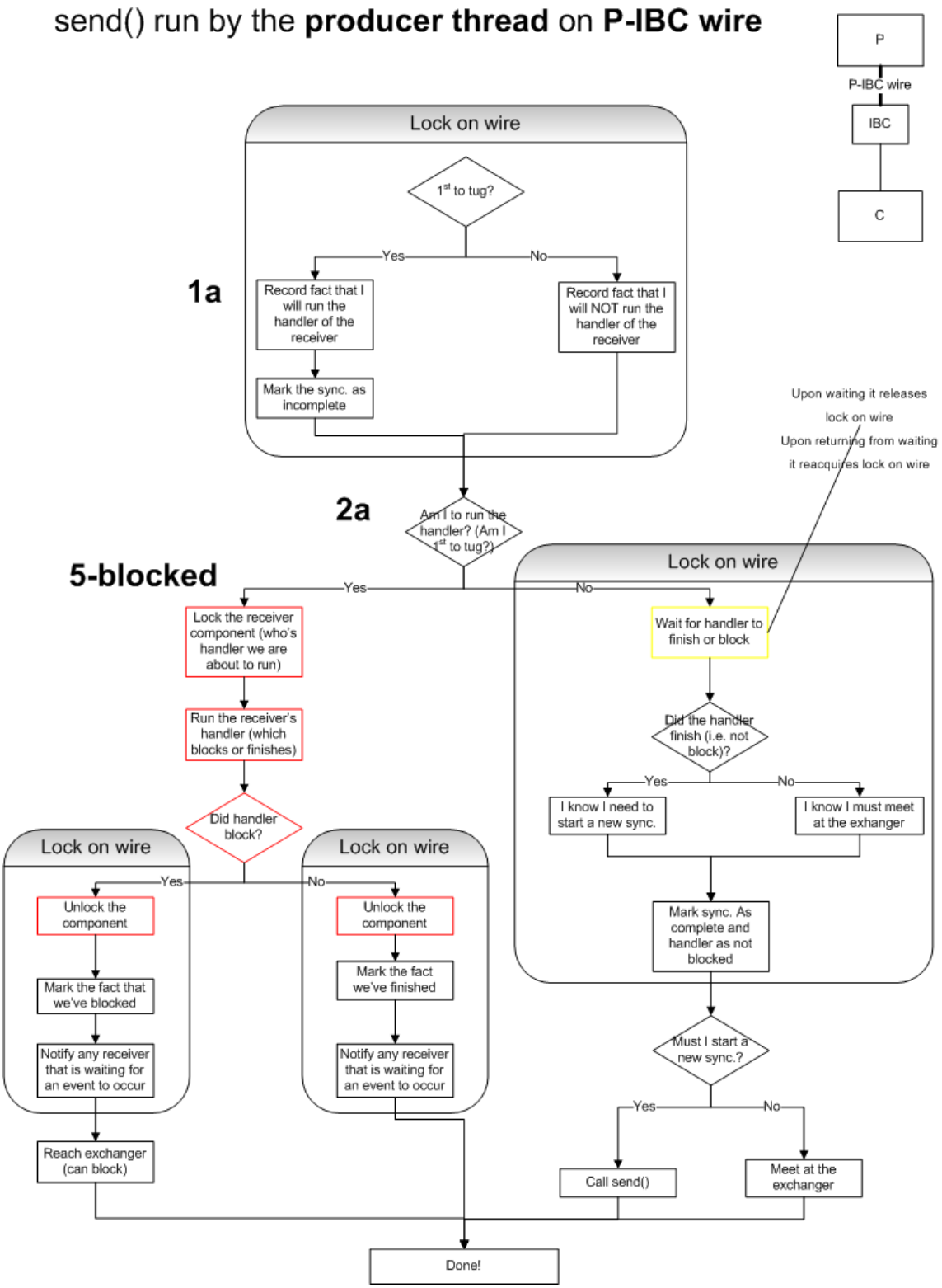


**Figure 61 - IntProducer, when executing its run method, wants to synchronise with IntBufferCell. Therefore send() is run by the producer thread on the Wire between IntProducer and IntBufferCell. The numbers shown indicate the order of execution by threads (producer thread in this figure) (letters are used when two events occur simultaneously).**

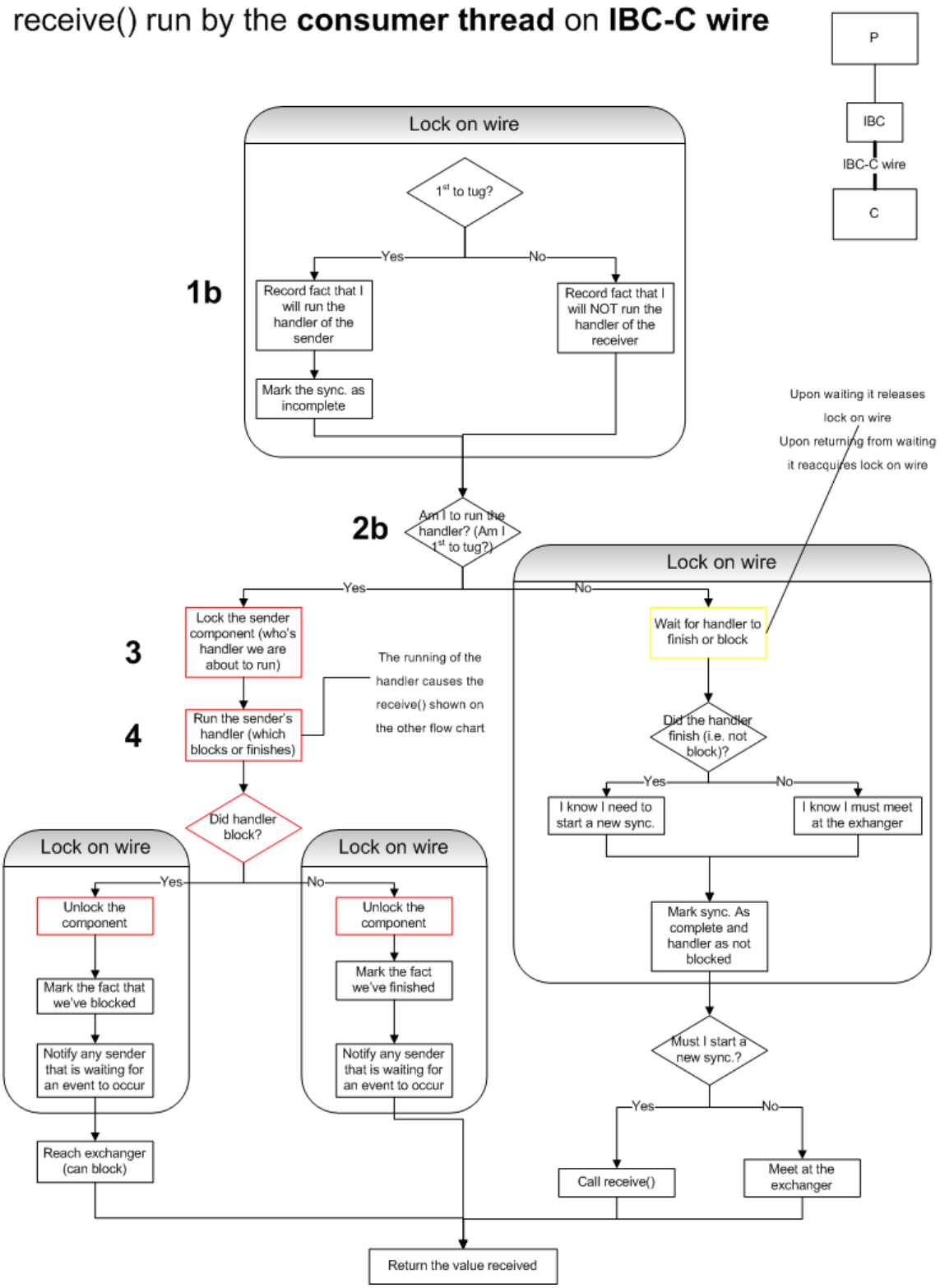


**Figure 62 - IntConsumer, when executing its run method, wants to synchronise with IntBufferCell. Therefore receive() is run by the consumer thread on the Wire between IntConsumer and IntBufferCell. The numbers shown indicate the order of execution by threads (consumer thread in this figure) (letters are used when two events occur simultaneously).**

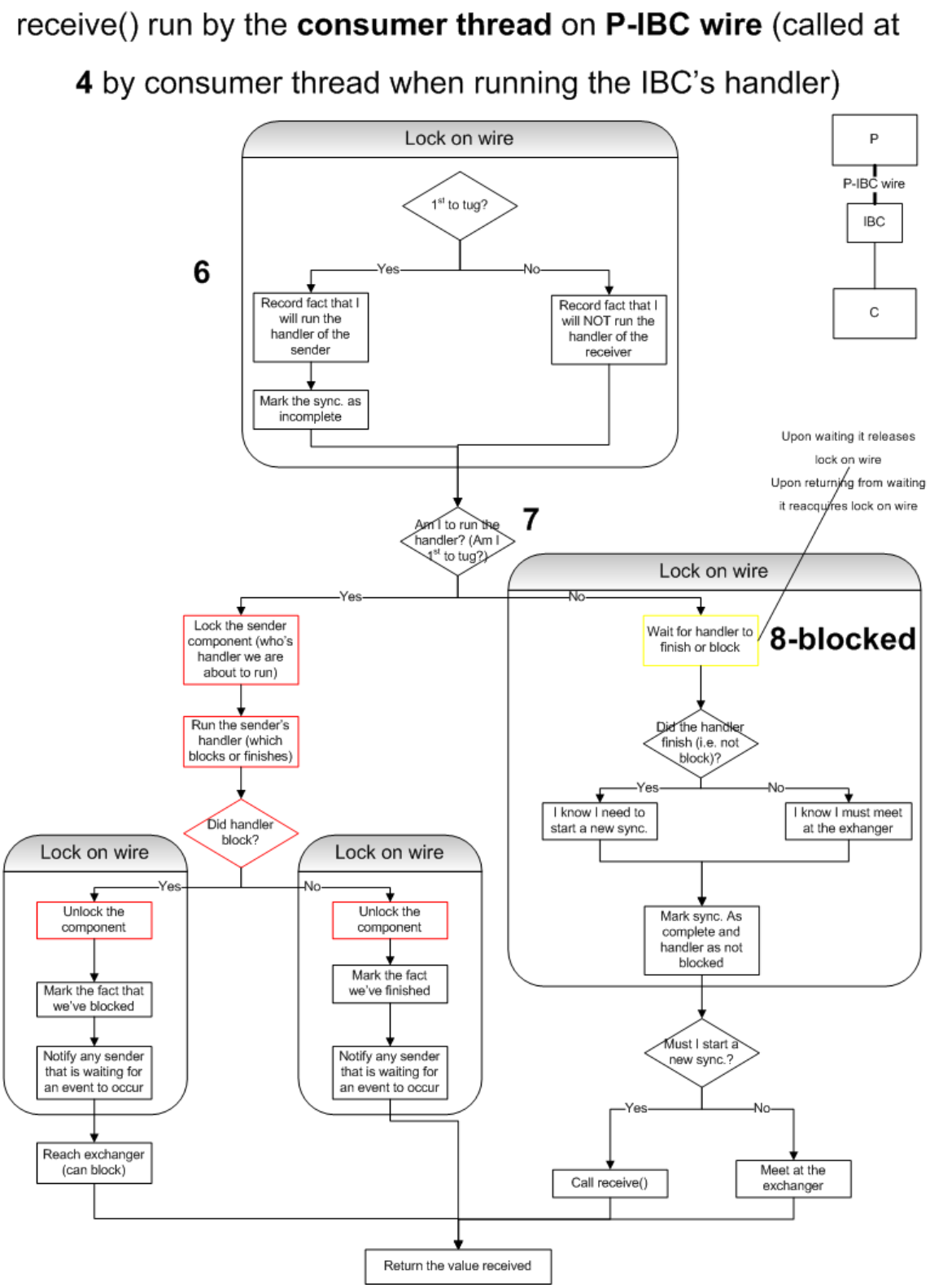


**Figure 63 - IntBufferCell, when executing its *out* handler, wants to synchronise with IntProducer. Therefore receive() is run by the *consumer thread (*who is running that handler) on the Wire between IntBufferCell and IntProducer. The numbers shown indicate the order of execution by threads (consumer thread in this figure) (letters are used when two events occur simultaneously).**

### F.3.3 Flow charts for Deadlock-Free NormalWire

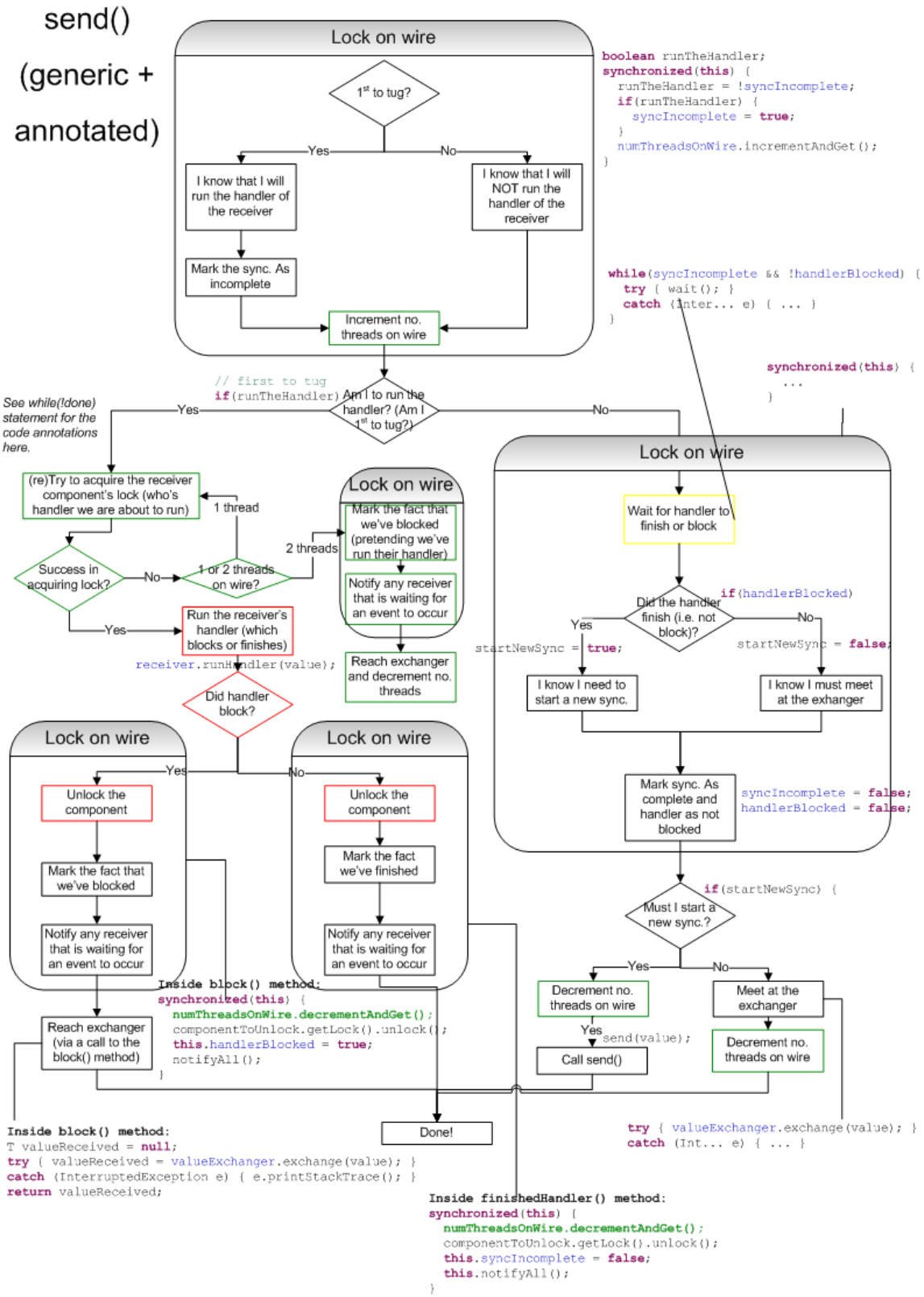


**Figure 64 - Flow chart showing logic of the corrected send() method of *deadlock-free* NormalWire, annotated with corresponding code fragments**

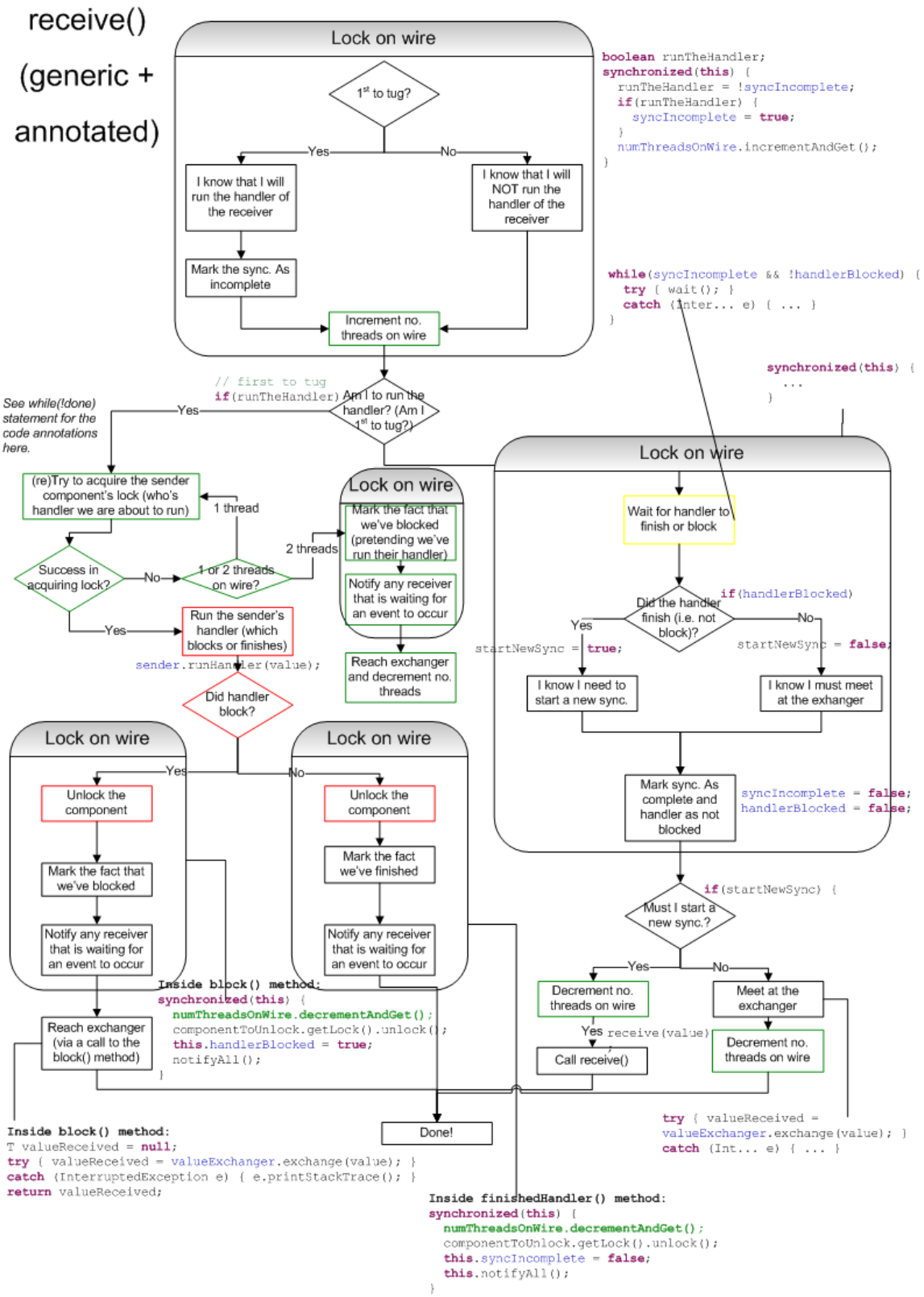


**Figure 65 - Flow chart showing logic of the receive() method of *deadlock-free* NormalWire, annotated with corresponding code fragments**

# Appendix G

# List of Semantic Checks

The following sections list the semantic checks performed.

## G.1 First Semantic Phase

### G.1.1 Component Definitions

**Component Definitions (in general)**

- SEMANTIC CHECK: checking that the names of components are distinct -- no two components can have same name/identifier
- SEMANTIC CHECK: warning (rather than an error) if there is component definition which has NEITHER a run method NOR any boundary declarations. Such a component would be useless (with no run method of its own and no boundaries, it cannot run of its itself nor can other components synchronise with it). Notify the programmer of this!
- (not implemented) SEMANTIC CHECK / TREE REWRITE: if there was is a boundary declared that has no corresponding handler then generate one for it (by rewriting the tree slightly).

**Boundary declarations**

- SEMANTIC CHECK: no two boundaries with same identifier (i.e. ensure this boundary identifier has not been used before). Boundary identifiers must be unique regardless of whether the rest of their signature is different (i.e. their types or direction).

**Run method declarations**

- SEMANTIC CHECK: ensure either zero (passive component) or one (active component) run method. No more than one run method is permitted.

**Handler declarations**

- SEMANTIC CHECK 1: this handler has not already been declared (i.e. at most a single handler can be declared per boundary)
- SEMANTIC CHECK 2: there should exist a boundary with the same name as the handler
- SEMANTIC CHECK 3: do the handler direction and type also match that of the boundary

**Outward synchronisation statements**

- SEMANTIC CHECK: identifier for boundary being sent on actually exists

**Inward synchronisation expressions**

- SEMANTIC CHECK: identifier for boundary being received on actually exists

### G.1.2 Wiring Code

**Composition Declarations**

- SEMANTIC CHECK: composition has not already been declared

**Composition Expressions** (reference to plain or composition component)

- SEMANTIC CHECK: an identifier in a composition expression is either a reference to a plain or composition component. Thus check there exists a plain or composition component with that identifier (by looking up in symbol tables)

**Start statement**

- SEMANTIC CHECK: composition to be started, represented by an identifier, refers to a composition that exists (i.e. there is a composition that has been declared with that identifier).

## G.2 Second Semantic Phase

The second semantic phase only performs wiring code checks. No further checks for component definitions is required. It ensures that the wiring specified by the programmer is valid.

### G.2.1 Wiring Code

**Composition Expressions**

- SEMANTIC CHECK: in sequential composition expression, ensure that the wirings between boundaries are all compatible.
- SEMANTIC CHECK: a reference to a previously declared composition component must be checked to ensure that the composition component has actually been declared.

**Start statement**

- SEMANTIC CHECK: ensure no remaining/'dangling' left or right boundaries of this composition component that is being started. Otherwise it is a type error.

# Appendix H

# Translator System Manual

The translator itself can be found on the attached DVD-ROM. The instructions here are based on the examples and directory structure used on the DVD-ROM. However, for these commands to run successfully, the DVD directories should be copied to another writable file system (since the DVD-ROM is read-only, the translator will fail if attempts are made to write the generated Java files to the DVD).

To simply run the pre-compiled translator without any rebuilding, then read only sections H.1 and H.2. To rebuild the translator from source then also see section H.3.

## H.1 Minimum System requirements

To simply *run* the pre-built translator on an input .javab or .javabc file:

- Java 6 **JDK** (see: http://www.java.com/en/download/help/sysreq.xml)
- ANTLR Parser Generator v3.3 (choose *Complete ANTLR 3.3 Java binaries jar (all tools and Java runtime)* at http://www.antlr.org/download.html); place this in a sensible directory (e.g. C:\ANTLR3.3\lib on Windows)
    - The CLASSPATH environment variable should be updated to include the path to the ANTLR runtime. Alternatively, Java's -cp option may be used instead.

## H.2 Running the translator

### H.2.1 Technical Restriction in Translating Wiring Code

To translate an application/wiring code file (.javab), the current implementation requires that all component definition files (.javabc) that are referenced in the wiring code file be translated in the same execution of the translator. Running the translator on the referenced components beforehand *in a different execution* of the translator will not help - all referenced components must be translated along with the wiring code file(s) *in the same execution of the translator*. Trying to do so will result in the translation failing.

This is a technical restriction due to the way the translator is implemented. Translating a single component definition file is allowable, however this is rarely of any use because any meaningful program will include wiring/application code file, and due to the technical restriction noted above, when that wiring code file is translated the component definition will also need to be (re)translated in the process.

### H.2.2 Running

The main translator program is JavaBTranslator.class. The translator can take several arguments, most of which are optional:

- -f or --files (compulsory)

Follow this option with the .javab and .javabc files to be translated

- -o or --output
  Output directory that the output from the translator should be put. If the directory does not exist, then it is created. If this option is not given then the current directory is used as the output directory.
- -a or --ast
  Setting this option is used for debugging purposes. When switched on, this option will print a string representation of the Abstract Syntax Tree (AST) produced by each internal phase of the translator.
- -p or --prettyast
  The same as -a/--ast but which uses the GraphViz *dot* program to produce a .png image of the AST. This option assumes that *dot* is installed on the system and also on the CLASSPATH. The .png image is placed in the output directory under a new directory 'ast-output'.
- -c or --javac
  Setting this option causes the translator to run the generated output (.java files) through javac, generating .class files. Follow this option with the path to the javab standard runtime library (which is inside /bin). If a following path is not given, the output directory will be used by default.

To run the translator for a real application, cd into /bin, and run:

```
java translator.JavaBTranslator -o ../translator-
generated/example_programs/02_P.IBC.C/ -f ../tests/example_programs/02_P.IBC.C/*
```

where the test example chosen might be different.

To translate and compile (using javac) in one command run:

```
java translator.JavaBTranslator -c ./ -o ../translator-
generated/example_programs/02_P.IBC.C/ -f ../tests/example_programs/02_P.IBC.C/*
```

To manually compile using javac, first cd into the directory containing the generated Java files, and run:

```
javac -cp .;path/to/javab/runtime/lib/ *.java
```

The classpath must be set to include the required Javab runtime library. To compile the example above would require (assuming the current working directory is translator-generated/example_programs/02_P.IBC.C/):

```
javac -cp .;../../../bin/ *.java
```

To then run the resulting Java program:

```
java -cp .;path/to/javab/runtime/lib/ Application
```

where Application is the name of the class that had the main() method within it. The directory containing the generated .class files is also specified in the classpath in addition to the path to the runtime library. Here the current directory contains the .class files. The above example would require:

```
java -cp .;../../../bin/ Application
```

## H.3 **Building from Source**

To build the translator from its source, one can build using Eclipse or manually.

To build the translator from within Eclipse, the ANTLRIDE plugin must be installed. The antlr-3.3-complete.jar file (ANTLR runtime) needs to be added to the build path of the project in order to do this. The ANTLR grammars can be built just by saving them. The translator's Java files generated by ANTLR are automatically compiled in/by Eclipse.

To build manually take the following steps:

### H.3.1 Generating the Parsers from Grammars

1. Navigate to the /src directory awhich contains the grammars and supporting Java classes.
2. First, run the ANTLR tool on the grammar files in that directory to generate the parsers written in Java:
   ```
   java org.antlr.Tool -o ../antlr-generated *.g
   ```
   This generates .tokens files in addition to the .java files, putting both into the /antlr-generated directory.

### H.3.2 Compiling the Parsers

3. In /src, copy JavaBTranslator.java into /antlr-generated
4. Navigate to the /antlr-generated directory.
5. Compile the ANTLR's generated .java files by running:
   ```
   javac -d ../bin -cp .;../src;C:/antlr-3.3/lib/antlr-3.3-complete.jar
   *.java
   ```

   Create the /bin directory if not done already.
   This assumes that the ANTLR jar is stored in the path used above.
   The -d option tells javac the directory to output all the .class files.
6. Finally, copy JavaBTemplates.stg from /src into /bin

The following webpages may be consulted for help when running java and javac on Windows:

- http://download.oracle.com/javase/1.4.2/docs/tooldocs/windows/java.html
- http://download.oracle.com/javase/1.4.2/docs/tooldocs/windows/javac.html

# Appendix I

# DVD Contents

/antlr-runtime
/JavaBTranslator
    /antlr-generated
        /translator
    /bin
    /src
        /javab
            /runtime
            /std_comp
        /junittests
        /translator
    /tests
    /translator-generated
    /yang-java-grammar
CONTENTS.txt
translatorSystemManual.pdf

The precise contents of the above directories is contained in the CONTENTS.txt file on the DVD-ROM.

The most important directory to note is /JavaBTranslator/src/translator which contains the translator source code files (all except JavaBTranslator.stg).

The /JavaBTranslator directory is also an Eclipse project and thus may be imported into Eclipse if desired. (To build the ANTLR grammars from within Eclipse, the ANTLRIDE plugin must be installed. The antlr-3.3-complete.jar file (ANTLR runtime) needs to be added to the build path of the project in order to do this).

translatorSystemManual.pdf contains a copy of Appendix H.