© Copyright by William Clarence Wendling, Jr., 2008

THAL++: A CONCURRENT LANGUAGE FOR THE THAL RUNTIME SYSTEM

BY

WILLIAM CLARENCE WENDLING, JR.

B.S., University of Illinois at Urbana-Champaign, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

ABSTRACT

THAL++ is a concurrent programming language that is a dialect of THAL. It principally differs from THAL in two ways: it has a syntax that is a subset of Java and it has types. Since the language is a subset of Java, THAL++ is easier to learn for programmers currently familiar with Java and Java-like languages. Other important features of the THAL++ compiler are primitives for asynchronous communication and concurrent call return communication (CC/RC) abstractions. These primitives make programming actor systems easier by allowing the programmer to focus on the algorithm rather than the details of communication and continuation.

To my parents, Bill and Carol Wendling. Much love!

ACKNOWLEDGMENTS

First off, this project would not have been possible without the support and patience of Prof. Gul Agha. He has been the calming force in a sometimes troubled sea.

I would not have been able to complete this project without the help and support of my friends: David, who proofread this thesis and offered advice; Geoff & Lesley, who have been great friends over the years; Wendy, my oldest and dearest friend; Randy & Kevin, for helping me see the bigger picture when things looked bleak; Chris & Tanya, for bolstering my confidence; and Carlos, for his love and for giving me a reason to finish this thesis.

I also wish to thank Café Kopi for many a night spent coding at their place and for the lovely teas which they serve.

TABLE OF CONTENTS

LIST OF FIGURES ix

CHAPTER 1 INTRODUCTION 1
1.1 The Actor Model of Computation 2
1.2 History of THAL++ 3
1.3 Overview 4

CHAPTER 2 BACKGROUND 5
2.1 Object-Oriented Programming 5
2.2 THAL 6
2.2.1 The THAL Language 6
2.2.2 The THAL Runtime System 7
2.3 Other Concurrent Object-Oriented Languages 8

CHAPTER 3 THE THAL++ LANGUAGE 10
3.1 THAL++ Design Choice 10
3.2 THAL++ Grammar 11
3.2.1 Comments 12
3.2.2 Identifiers 12
3.2.3 Using 12
3.2.4 Types 13
3.2.5 Method Prototypes 13
3.2.6 Interfaces 14
3.2.7 Methods 15
3.2.8 Statements 16
3.2.9 Classes 18

3.3 Transformations 21
3.3.1 Actor Creation Transformation 21
3.3.2 Asynchronous Message Passing Transformation 22
3.3.3 CC/RC Transformation 22

CHAPTER 4 EXPERIMENTS AND RESULTS 27
4.1 Fibonacci Number Generator 28
4.2 N-Queen Problem 30

CHAPTER 5 CONCLUSION 34

REFERENCES 35

APPENDIX PROGRAMMING FOR THAL'S RUNTIME SYSTEM 37
A.1 Global Variables 37
A.2 Global Structures and Enumerations 39
A.3 Transforming THAL++ Methods Into C Functions 41
A.3.1 Calling Conventions 41
A.3.2 Code Transformations 42

VITA 46

LIST OF FIGURES

Figure **Page**

1.1 Graphical representation of an actor system. 3

3.1 The program uses the C method `printf`. 13
3.2 Declaration of the public method `Compute`. 14
3.3 `Algorithms` interface declaring the `Compute` method. 14
3.4 Implementation of `Compute` method that computes the factorial of an integer. 15
3.5 The `new` operator creating a `Factorial` object on node 3. 17
3.6 Sending an asynchronous message to an object. 19
3.7 Computing and printing the factorial of 9 via CC/RC. 20
3.8 Declaration of class `Factorial` which can now be used as a type. 20
3.9 Sending a message to a temporary actor. 22
3.10 Transformation of a CC/RC message into an asynchronous message send in straight line code. 23
3.11 Transformation of a CC/RC message into an asynchronous message send in an `if-then` statement. 24
3.12 Transformation of a CC/RC message into an asynchronous message send in loop. 26

4.1 THAL++ program to compute the Fibonacci of a number on `NumProcs` nodes. 28
4.2 Graph of the computation time for calculating *Fibonacci(N)*, where $N = 0, 1, 2, \ldots, 30$. 29
4.3 Part 1 of a THAL++ program to compute the number of solutions for placing N queens on an $N \times N$ board. 30
4.4 Part 2 of a THAL++ program to compute the number of solutions for placing N queens on an $N \times N$ board. 32

4.5 Graph of the computation time for calculating the solution to the N-Queens problem for $N = 0, 1, 2, \ldots, 12$ 33

A.1 Program which has three message sends. 39
A.2 Example entry into the `_method_table_entry` array. 40
A.3 C code for creating an actor. 43
A.4 C code for sending an asynchronous message to an actor. 44
A.5 C code for filling a continuation. 45
A.6 C code for returning from a function that is the result of a CC/RC call. . . . 45

CHAPTER 1

INTRODUCTION

The advent of cheap and powerful multiprocessor machines and clusters has given us the ability to perform calculations that would otherwise take too long to perform on single processors or would be cost-prohibitive. Traditional, non-concurrent programming languages and models of computation—such as sequential processes and functions transforming data values—can be difficult to program for these environments and aren't as powerful as the actor model. The programmer in such an environment needs to handle communication, synchronization, and scheduling for any number of processes. The task is quite complex and difficult.

The actor model, proposed by Carl Hewitt in the 70's and refined by Agha[1] in the 80's, fits naturally into a parallel environment. An actor, which can be thought of as an object which responds to messages, resides on a *node* in the environment. Actors are autonomous: each actor knows only what types of messages it or other actors can accept. Messages are routed via a communications substrate. This substrate either knows the distribution of actors on the network or can quickly determine the position of an actor within the environment.

An actor may migrate from one node to another if it's determined that this would benefit overall performance. In this manner, actors are able to exploit parallel environments.

1.1 The Actor Model of Computation

The actor model of computation is an abstraction of a concurrent runtime environment. Actors are self-contained objects which have their own internal states and threads of control. Actors can subsume both sequential processes and value-transforming functional systems. "It is possible to define a purely functional system as an actor system and it is possible to specify arbitrary sequential processes by a suitable actor system, but it is not possible to represent an arbitrary actor system as a system of sequential processes or as a system of value-transforming functions."[1] Therefore, an actor system is more powerful than either a sequential processing or value-transforming functional system.

Actors communicate via message passing to other actors they know about. The communications substrate is responsible for finding the receiving actor and sending the message to that actor. Messages for an actor are placed in that actor's *mail queue*. Messages are acted upon in the order they are received in the queue, and, thus, no assumptions can be made about what order messages will be acted upon by an actor. In response to a received message, the actor can do several things. It may create a finite set of actors, send a finite set of messages to other actors it knows about, change its local state to become ready to process another message, or a combination of these actions.[1] Figure 1.1 shows a graphical

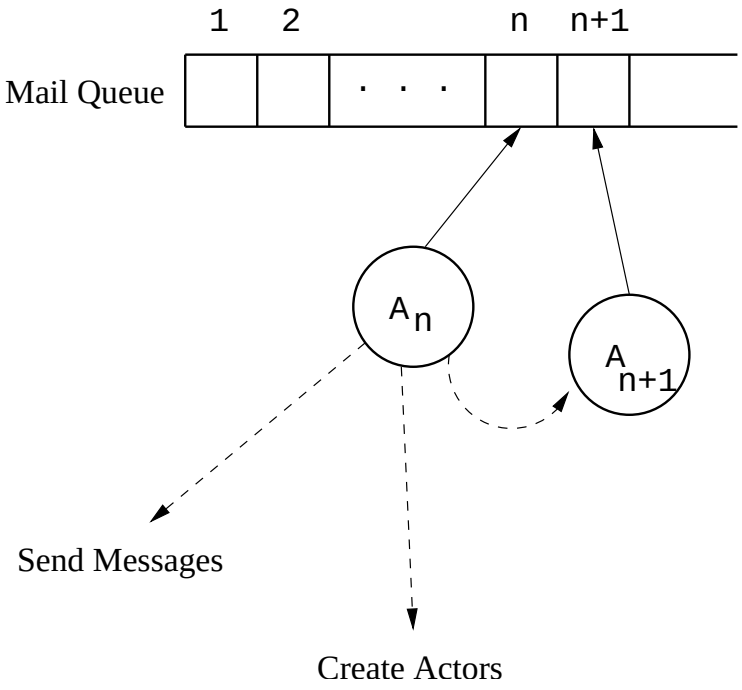

Figure 1.1 Graphical representation of an actor system.

representation of an actor in an actor system, where A_n is an actor after it has read message n and A_{n+1} is that same actor after it has read message $n+1$.

Message passing in actor systems is *asynchronous*. Once a message is sent, the actor sending the message resumes execution of the method directly after the message send point without waiting for a response. A *synchronous* message send can be simulated in actor systems via the *concurrent call/return communication* abstraction (see Section 2.2.1).

1.2 History of THAL++

THAL++ is modeled after THAL, developed by Wooyoung Kim[2]. THAL is a concurrent object-oriented programming language and runtime system that supports the actor

model of computation. The language provides abstractions for programming actors while the runtime system provides an efficient implementation of the message passing required for actor programming.

1.3 Overview

This thesis is organized as follows: Chapter 2 gives a background of THAL, the Actor Foundry, and other concurrent object-oriented programming languages. Chapter 3 describes THAL++'s syntax. Chapter 4 shows some programs written in THAL++ and how they perform. Chapter 5 wraps it all up with a conclusion.

CHAPTER 2

BACKGROUND

In this chapter, we present an overview of object-oriented programming, how actor languages benefit from it, an overview of THAL, and a survey of some concurrent object-oriented languages.

2.1 Object-Oriented Programming

One of the key tenets of the object-oriented paradigm is the idea of encapsulation. An object encapsulates internal information—state variables—which can be accessed only via well defined interfaces (methods). Communication between objects is abstracted to invoking a method on an object. Thus, each object is independent and knows only which methods other objects can accept.

Through inheritance, an object of type D which is derived from type B may be treated as if it is an object of type B. Therefore, given an object of type B and a method name M, the actual method being invoked may vary depending upon the actual type of the object. In general, given an arbitrary object of type D and a method name M, it is impossible to determine statically (i.e., at compile time) which method will be invoked. However, it is

possible to determine if an object of type D can accept a method name M. This allows for flexibility in the language. Actors of heterogenous types can be treated in the same fashion if they have the same base type or implement the same interface. The actual method invoked is determined dynamically at runtime.

Concurrent object-oriented languages use this paradigm and add a separate thread of control for each object. Method invocations are done via message passing.

2.2 THAL

THAL[1] was developed by Wooyoung Kim. It is a high level language for developing actor programs. The THAL language supports grouping and communication abstractions and performs automatic garbage collection. The runtime system makes the message passing between actors efficient.

2.2.1 The THAL Language

THAL has an object-oriented syntax. Actors are behaviors (objects) that have associated methods and internal state variables. Methods are the target of message sends. Types of variables are inferred by the compiler and runtime system.[2]

THAL has the concept of *group abstraction*. The ability to treat actors of the same type as a group gives the programmer a simple and efficient way of having all the actors in the same group acting upon the same message. Actors in a group can be treated as a single actor.

[1] "THAL should be pronounced as [tal]. It is a Korean term ... for masks that are used in a traditional Korean dance-and-play which is loved and enjoyed especially by common folks."[2]

Individual actors in a group all receive a message sent to the group via group communication abstractions. Membership within the group can change dynamically, something which would be difficult to program without group abstractions.[2]

THAL supports communication abstractions such as concurrent call/return communication (CC/RC) in which the program sends a message to an actor and expects a response before it continues code execution. Delegation is supported, so an actor can specify that the response to a message should be sent to an actor other than itself. Local synchronization constraints allow for an actor to accept a message only when it is in a state in which it is able to accept the message. Actors exist on nodes which have unique numbers associated with them. Actors can be placed on different nodes dynamically via annotations when creating the actor (i.e., creating the actor *ActorName* on node *NodeNumber* using the syntax "`new` *`ActorName`* `on` *`NodeNumber`*") or via migration.[2]

2.2.2 The THAL Runtime System

The THAL runtime system is a virtual machine located between the actor and the communications layer. It is in charge of making sure an actor receives the messages intended for it.

The runtime system is comprised of various parts, described in [2]:

- *Node Manager* — The node manager is in charge of delivering messages, creating actors on remote nodes, and dynamical loading of user code into the system.

- *Dispatcher* — The dispatcher provides the framework for actor scheduling. The actual scheduling is left to the individual actors.

- *Name Server* — The name server translates mail addresses to physical locations.

It is the responsibility of the compiler to try to eliminate message send calls when possible. When an actor is determined to be local to the processor, then the message send can be transformed into a local function call, requiring much less overhead.

2.3 Other Concurrent Object-Oriented Languages

Cantor[3] is one of the first actor languages. It deviates from the actor paradigm by assuming that two messages sent from one actor will arrive in the order they were sent, an assumption actor systems generally do not rely on. Cantor is a very basic language. Iteration (i.e., looping) is not one of the basic actions. Nor are common structures such as arrays or records available. These must be programmed using Cantor's objects.

ABCL/1[4] is an object-oriented concurrent programming language that makes extensions to the basic actor model. Like Cantor, ABCL/1 assumes that multiple messages sent from one actor to another are totally ordered. Objects can be in one of three modes: *dormant*, *active*, or *waiting*. An object is dormant when its message queue is empty, it is active when processing a message, and it is waiting if it needs information not currently available to it. When in waiting mode, messages can still be processed. There are two different message queues: *ordinary* and *express*. Express messages are processed immediately if the object is currently either processing an ordinary message or is in a dormant state. ABCL/1 introduces

three different ways of sending a message: *past*, *now*, and *future*. *Past* and *future* messages are asynchronous, while *now* messages are synchronous.

The Actor Foundry[5] was developed by the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. It is an actor runtime system developed in Java. Actor programs are written in Java, while the foundry provides an architecture for managing actors, a name service to find where an actor is located, a transport layer to pass messages to actors, and a scheduler to schedule all of the threads in the system.

Concurrent Aggregates (CA)[6] is an object-oriented concurrent language which uses a hierarchy of abstractions to provide more concurrency than available from the actor model. Objects are placed in aggregates each of which is treated as an object. A message sent to the aggregate will go to an arbitrary object within that aggregate. Multiple messages can be handled in this manner, unlike in the actor model where messages to an object are processed in a serial fashion.

SALSA[7] is an actor-oriented programming language that compiles into Java code. It simplifies actor-oriented programming and worldwide computing, which is a project designed to turn the world wide web into a unified, dependable, distributed computing environment. It provides a universal naming scheme for actors, asynchronous communication, active objects, and synchronization through continuations.

CHAPTER 3

THE THAL++ LANGUAGE

Creating a new language is necessary only when it provides useful abstractions that are difficult or impossible to implement in another language. Usually, a new language either borrows heavily from current, popular languages (like C++ or Java) or else is an extension to a language it wishes to enhance (like C++ is to C).

The main goal of creating the language THAL++ is to alleviate some of the problems that THAL has: a syntax unlike current, popular languages such as Java and C++; and the lack of a type system.

3.1 THAL++ Design Choice

There were three choices for designing THAL++: extending an existing language, such as Java, to accept some actor message passing primatives and run on THAL's runtime system; modify THAL to have types and a syntax more like C++ or Java; or create a new language. Each choice has its own strengths and weaknesses.

Extending an existing language has the benefit that the language is already potentially in wide use and there already exists a compiler, obviating the need to write a new compiler

from scratch. The drawback of extending a current language is that the language needs to interface with the runtime system. This means that all of the data structures the language supports need to be either adapted into the runtime's message passing system or left out. But supporting all the types of data structures in a language, like Java, is a large task (beyond the scope of this project) and would force changes in the runtime system.

Modifying THAL has the advantage that THAL already has a compiler. However, the changes needed to the language are so extensive that they would involve a complete rewrite of the THAL compiler. All aspects of the compiler are touched by such changes, from the front end to the intermediate representation to the back end.

In light of the problems of the first two choices, the decision was made to create the new language THAL++. It has a syntax similar to Java, a type system, and supports some actor message passing primitives, although it isn't as full-featured as Java. This chapter assumes that the reader has a working knowledge of Java.

3.2 THAL++ Grammar

This section describes THAL++'s grammar. It should be familiar (if limited) to someone who has a working knowledge of Java. Therefore, much of the grammar—e.g., description of `for`, `if-then`, `while`, and `do-while` statements—is not specified here since they have the same syntax as in Java. We note the places where THAL++'s expressions differs from Java expressions.

3.2.1 Comments

THAL++ supports Java-style comments. A block of text is commented using `/* ... */`. For a single line comment, the programmer can use `//`, which comments the entire line until a newline character is read. Note that the `/* ... */` commenting method doesn't nest, while the `//` commenting method does.

3.2.2 Identifiers

An identifier in THAL++ must begin with either an underscore (_) or alphabetic character. After the initial character, there may be zero or more alphanumeric characters or underscores. For example, `Actor_1` is an acceptable identifier but `1_Actor` isn't. THAL++ is a case-sensitive language. Therefore, the identifiers `myID` and `MyID` are two different identifiers.

3.2.3 Using

The `using` keyword is used to declare external C methods used in a program. Only the name of the method should be specified. A comma separated list of C method names may be specified. All external C methods used in the program must be specified with a `using` statement and the declaration must come before any use of that method. The `using` statement must be followed by a semicolon. Figure 3.1 indicates that the program uses the C method `printf`.

```
using printf;
```

Figure 3.1 The program uses the C method `printf`.

3.2.4 Types

THAL++ has the following built-in types: boolean, character, integer, float, string. The keywords for each built-in type are `bool`, `char`, `int`, `float`, and `string` respectively. The sizes in bytes of `bool`, `char`, `int`, and `float` are 1, 1, 4, and 8 respectively. The size of `string` is dependent upon the length of the string. Except for `bool` and `string`, the types are all signed.

In addition, the names of interfaces and classes can be used as types. Specifying an array is done as, in C, by specifying the size in brackets after the identifier. For example, `int a[10];` declares an array of 10 integers. The size must be a constant integral value when declaring the array type and must not be omitted.

Initialization of variables in type declarations are not permitted. Variables can only be declared at the beginning of a block of statements—a set of statements beginning with a `{` and ending with a `}`, as in the true and false parts of an if-then statement and in the bodies of loops.

3.2.5 Method Prototypes

Method prototypes are used in interfaces and specify the public methods a class must define if it implements an interface. Method prototypes consist of the method return type,

name, and parameter types the method must have. All method prototypes have the default scope of "public," so it is not necessary to specify the scope explicitly. The syntax for method prototypes is similar to method prototypes in Java. Figure 3.2 is the method prototype for a *Compute* method that takes no arguments and returns an integer value after running an algorithm.

```
int Compute();
```

Figure 3.2 Declaration of the public method `Compute`.

3.2.6 Interfaces

An interface is used to specify the methods that a class which implements the interface must define. These methods are public by default, which means that another actor can invoke the method via message sending. Interfaces in THAL++ are similar to interfaces in Java: the body of the interface is a list of method prototypes. Classes which extend the interface are type compatible. Figure 3.3 shows the interface named *Algorithms* which declares the method *Compute* from Section 3.2.5.

```
interface Algorithms {
  int Compute();
}
```

Figure 3.3 `Algorithms` interface declaring the `Compute` method.

3.2.7 Methods

A method can be either *public* or *private*. Public methods are accessible by other actors via message passing. Private methods are accessible only by the actor which defines the method and are called without message passing. Constructors are special methods. They must be declared public, have no return type, and have the same name as the class in which they're defined. There can be at most one constructor for a given class.

Methods resemble Java methods. They have variable declarations at the beginning of blocks of statements and a list of statements which are executed in order. The statements `if-then-else`, `for`, `do-while`, and `while` loops are the same as in Java.

Figure 3.4 shows an implementation of the *Compute* method from Section 3.2.5. It calculates the factorial of the value the class was initialized with via the constructor.

```
public int Compute() {
  int result, n;

  n = Number;

  if (n <= 1)
    return n;

  for (result = 1; n > 0; n -= 1)
    result *= n;

  return result;
}
```

Figure 3.4 Implementation of `Compute` method that computes the factorial of an integer.

3.2.8 Statements

Statements in THAL++ are the same as those in C with only a few additions and subtractions. To support concurrency, THAL++ introduces the `new` operator (see Section 3.2.8.2), the asynchronous message passing operator (see Section 3.2.8.3), and the concurrent call/return communication (CC/RC) operator (see Section 3.2.8.4). The C expressions THAL++ doesn't support are listed in Section 3.2.8.1.

3.2.8.1 Unsupported Java Features

The ternary (`?:`) and unsigned right shift (`>>>`) operators are not supported. Resources were not available to support these operators. In addition, `switch`, `continue`, and `break` statements aren't supported. These statements are mostly "syntactic sugar" and can be coded in other ways. The member access operator (`.`) has a different meaning in THAL++ (see Section 3.2.8.4). As in Java, THAL++ doesn't support `goto` statements.

3.2.8.2 Actor Creation

To create an actor, the programmer uses the `new` operator. The syntax for the `new` operator is

`new` *ClassName*`(...)` [`on` *Expression*]

where *ClassName* is the name of the class being instantiated and "`on` *Expression*" is optional. *Expression* must evaluate to an integer value which is the node number in the system. If "`on` *Expression*" is omitted, the actor is created locally.

Figure 3.5 shows a program which creates a *Factorial* object on node 3 using the `new` operator. Notice that there isn't a `delete` operator for actors. This is intentional; THAL++ relies upon the THAL runtime system to perform garbage collection of actors.

```
interface Algorithms {
  int Compute();
}

class Factorial implements Algorithms {
  int Number;

  public Factorial(int n) { Number = n; }

  // Implementation of Compute function...
}

class Main {
  public void main() {
    Factorial Fact;

    Fact = new Factorial(9) on 3;
    // More code...
  }
}
```

Figure 3.5 The `new` operator creating a `Factorial` object on node 3.

3.2.8.3 Asynchronous Message Passing

Messages are passed to an actor using the asynchronous send (`<-`) operator. Execution of the method continues immediately after the message is sent without waiting for a reply. The left hand side of the operator is an actor created via the `new` operator (see Section 3.2.8.2). The right hand side of the operator is the name of a public method in that actor and a comma separated list of arguments within parentheses. THAL++ sends the message via the THAL runtime system to that actor and expects no return value. Program execution continues

immediately after the message is sent. Figure 3.6 shows a send of the message `Compute` to a `Factorial` actor; the resulting output will read, "`Factorial of 9 is 362880`."

3.2.8.4 Concurrent Call/Return Communication (CC/RC)

Concurrent call/return communication (CC/RC) is a convenient way of programming synchronous message passing in the actor model, which only allows for asynchronous message passing. The member access (.) operator is used to indicate a CC/RC call. For instance, in the Factorial example in Figure 3.5, we would like to compute the factorial and print the result afterwards. Figure 3.7 shows a message being sent to the actor telling it to compute the factorial with the subsequent printing of the result of that computation.

3.2.9 Classes

A class defines an actor. It may optionally implement one or more interfaces. The body of a class includes its class variables, the methods the actor defines (see Section 3.2.7), and an optional constructor. The constructor, if present, must have the same name as the class and be public. There can only be one constructor per class. Class variables may be declared. They are accessible only to public and private methods within that class so they must not have a scope signifier. The programmer may initialize them via the constructor or public accessor methods. Public methods can be targets of message sends (see Sections 3.2.8.3 and 3.2.8.4) while private methods are accessible only to methods within the class in which they are defined.

```
using printf;

interface Algorithms {
  void Compute(int N);
}

class Factorial implements Algorithms {
  public void Compute(int N) {
    int prod;

    if (N == 0) {
      prod = 0;
    } else if (N < 0) {
      printf("Cannot take factorial of negative number\n");
      return;
    } else {
      for (prod = 1; N > 0; N -= 1)
        prod *= N;
    }

    printf("Factorial of %d is %d\n", N, prod);
  }
}

class Main {
  public void main() {
    Factorial Fact;

    Fact = new Factorial();
    Fact <- Compute(9);
  }
}
```

Figure 3.6 Sending an asynchronous message to an object.

```
interface Algorithms {
  int Compute();
}
class Factorial implements Algorithms {
  int Number;

  public Factorial(int n) { Number = n; }

  // Implementation of Compute function...
}

class Main {
  public void main() {
    Factorial Fact;

    Fact = new Factorial(9);
    printf("The factorial of 9 is %d\n", Fact.Compute()); // CC/RC call
    // More code...
  }
}
```

Figure 3.7 Computing and printing the factorial of 9 via CC/RC.

A class must be either declared or defined before it can be used as a type. Figure 3.8 shows how to declare a class. After the declaration of `Factorial` the programmer may now use `Factorial` as a type. A class definition must follow before the end of the program.

```
class Factorial;
```

Figure 3.8 Declaration of class `Factorial` which can now be used as a type.

The method name `main` is reserved and is the starting place of the program. It should be declared `public`, have a return type of `void`, and take no arguments. There must be one and only one definition of the method `main`.

3.3 Transformations

The THAL++ compiler converts the code into C code which is then compiled against the THAL runtime library. Class variables and methods are global in scope. The namespace is kept ordered by prepending the class name before the variable and method names.

Messages sent to an actor have their parameters placed into an `ActorMessage` structure. This structure is the argument used when invoking the corresponding method. Therefore, each public method has an entry method which receives this `ActorMessage` structure, takes the parameters from within this structure, and passes them the implementation of the method.

3.3.1 Actor Creation Transformation

If the actor is locally created (i.e., the "`on` *Expression*" syntax isn't used), the actor is allocated locally and then calls the initialization method (constructor) with the parameters, if any, given to the constructor.

If the optional "`on` *Expression*" syntax for the `new` expression is used, then a message is sent to the node where the actor is to be created. An *alias* is used on the local node so that, once the message is sent, the creation method can return. The program then uses this alias as if it were the actor. The runtime system is in charge of coordinating the creation of the remote actor and the messages being sent to it.

3.3.2 Asynchronous Message Passing Transformation

An asynchronous message passing statement has to create an `ActorMessage` structure, assign the parameters to the correct places in the structure, and then send this message to the actor in question. It is possible to create an actor and send a message to it if that actor isn't to be used again. Figure 3.9 shows how to send a message to a new actor that will be local on the current processor.

```
new Foo() <- Message(param1, param2);
```

Figure 3.9 Sending a message to a temporary actor.

3.3.3 CC/RC Transformation

A CC/RC message requires several bits of analysis before it can be translated into proper C code. The compiler performs live variable analysis and def-use chains analysis for each method using the algorithms from [8]. Def-use chains tell us where a CC/RC result is used. Live variable analysis tells us what values are needed at the point of a CC/RC call.

This information is needed because at the point of a CC/RC call we are going to convert the CC/RC into an asynchronous message send. After all of the actors receiving these asynchronous messages are finished processing the messages, the code dependent upon the value of those computations is called. This code is placed into another method called a *continuation*. The continuation receives a message containing the required values from the

message sends along with any variables which may have been alive at the point the CC/RC was called.

The transformations from CC/RC calls to asynchronous message sends for the various types of code is modeled after how THAL performs these transformations in [2].

For straight line code, the code is split at the point of a CC/RC call. The CC/RC call is converted into a message send and the code after the CC/RC call is placed into a continuation. Figure 3.10 shows how straight line code is converted.

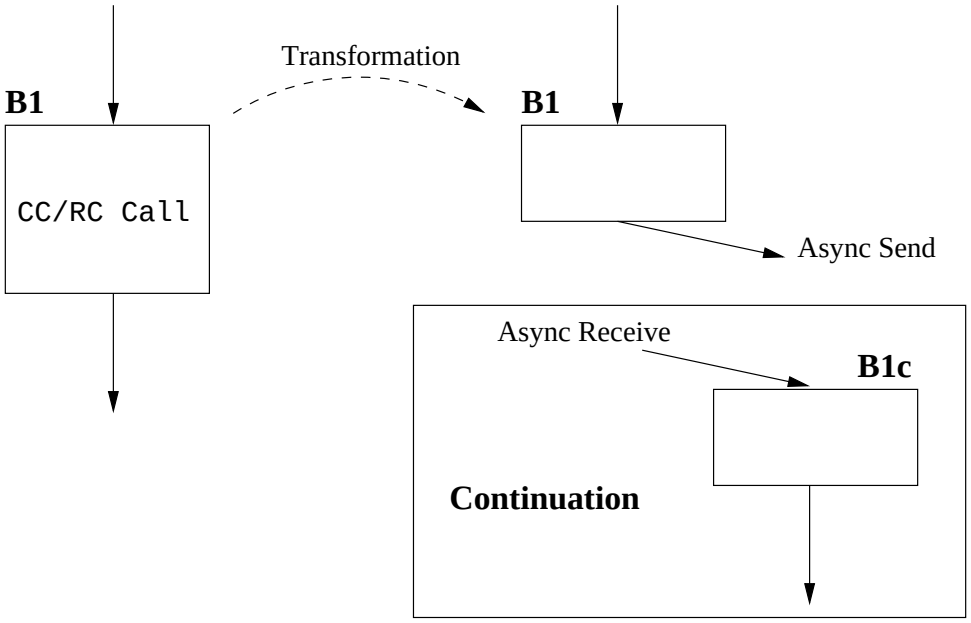

Figure 3.10 Transformation of a CC/RC message into an asynchronous message send in straight line code.

For an if-then statement, the branch which contains the CC/RC call is split as with straight line code; however, all of the statements after the if-then statement are copied into the continuation method. Figure 3.11 shows how an if-then statement is converted.

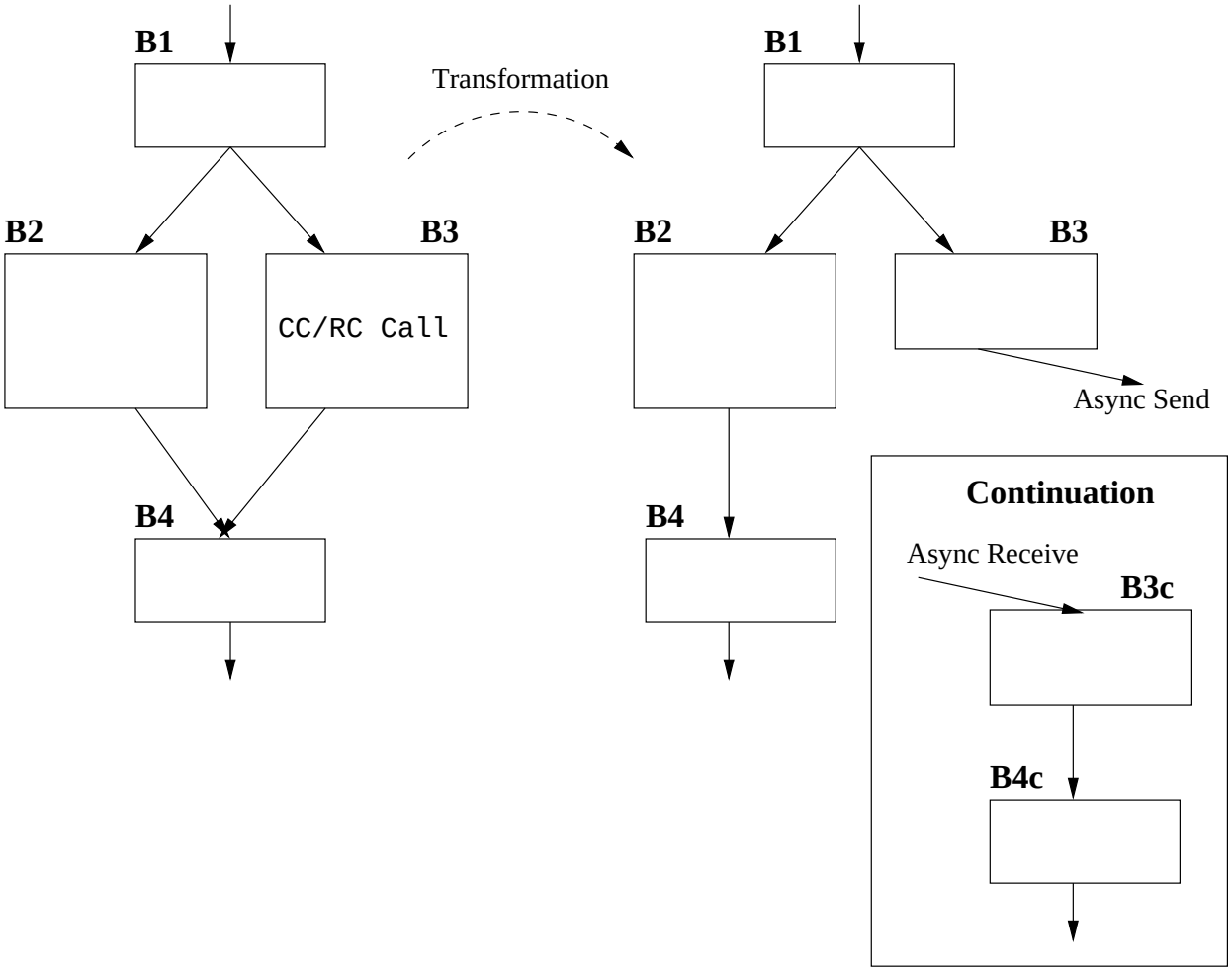

Figure 3.11 Transformation of a CC/RC message into an asynchronous message send in an `if-then` statement.

Loops with CC/RC calls are more difficult to transform than either straight line code or if-then statements. The loop statement is transformed into an if-then statement. Figure 3.12 shows how the loop is transformed into an if-then statement with a continuation. The conditional expression for the loop is used as the conditional for the if-then statement. The body of the loop is split so that the part before the CC/RC call is left intact. An asynchronous message send is inserted for the CC/RC call. The rest of the body is the beginning code for

the continuation method. The if-then statement is then replicated within the continuation method.

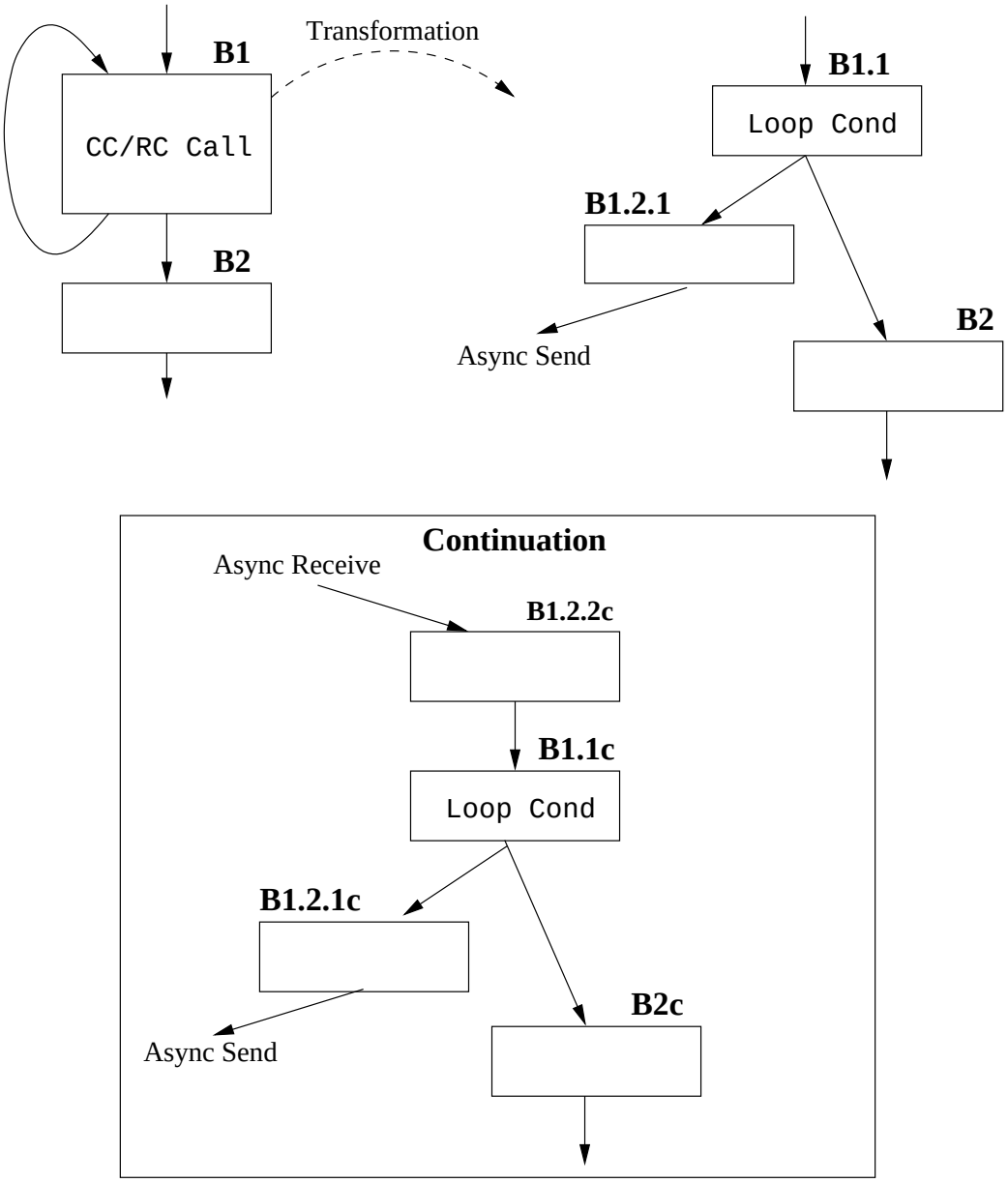

Figure 3.12 Transformation of a CC/RC message into an asynchronous message send in loop.

CHAPTER 4

EXPERIMENTS AND RESULTS

In this chapter, we present programs written in THAL++ and the timing results of the programs with different inputs. These programs are designed to be distributed on the nodes of a system in a balanced way.

The programs are compiled using the THAL++ compiler. The result is a C program file that has the same name as the original file with `___THAL.c` appended to it. This file is then compiled using the GNU Compiler Collection's (GCC)[1] C compiler. The executable is linked with THAL's runtime library. Currently, the THAL runtime system runs on Microsoft® Windows 2000™ running Cygwin™[2]. THAL is available from the Open System Laboratory (OSL) at the University of Illinois at Urbana-Champaign. See `http://osl.cs.uiuc.edu/` for more information about THAL.

Note: Unfortunately, at the time of the writing of this thesis, the THAL runtime system is only able to run on one node. Therefore, the timing results are for a single processor machine. After the runtime system is working sufficiently for multiple nodes, we will present more results.

[1] See `http://gcc.gnu.org/` for more information about GCC.
[2] See `http://www.cygwin.com/` for more information about Cygwin.

4.1 Fibonacci Number Generator

The Fibonacci number generator returns the Fibonacci number for the number n, where the $Fibonacci(n)$ is determined by the recurrance relation: $Fibonacci(0) = 1$, $Fibonacci(1) = 1$, ..., $Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)$. The recursive algorithm used in this experiment has an exponential time complexity. The reason for using the recursive algorithm here is to demonstrate how well the THAL runtime system handles communication, since calculating $F(n)$ requires X messages.

Figure 4.1 shows the code for the Fibonacci program. The program balances the computations among the processes by having node P compute the Fibonacci of Q mod $NumProcs$, where $NumProcs$ is the number of processes in the system.

```
using printf;

class Fib {
  int NumProcs;
  public Fib(int nprocs) { NumProcs = nprocs; }
  public int compute(int N) {
    if (N <= 1) {
      return 1;
    } else if (N == 2) {
      return 2;
    } else {
      Fib f1, f2;
      f1 = new Fib() on (n - 1 % NumProcs);
      f2 = new Fib() on (n - 2 % NumProcs);
      return f1.compute(n - 1) + f2.compute(n - 2);
    }
  }
}
```

Figure 4.1 THAL++ program to compute the Fibonacci of a number on `NumProcs` nodes.

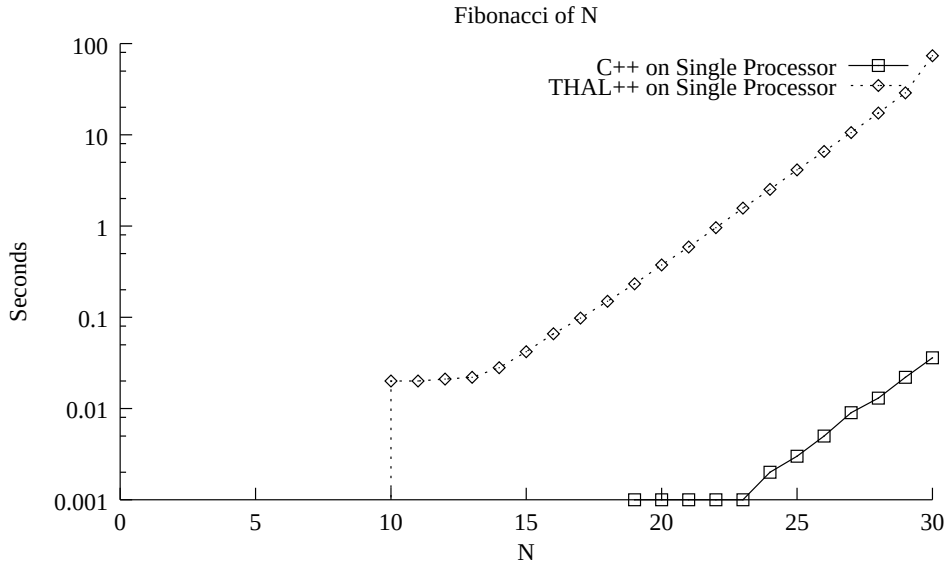

Figure 4.2 Graph of the computation time for calculating *Fibonacci(N)*, where $N = 0, 1, 2, \ldots, 30$.

Figure 4.2 shows the results of calculating the Fibonacci sequence on one processor. The C++ program uses a recursive algorithm and has an exponential time complexity. The THAL++ program is a modified version of the program in Figure 4.1. It doesn't allocate actors on remote nodes. A logarithmic graph better illustrates the differences between the run-time of the C++ and THAL++ programs.

As the graph shows, the time the THAL++ program takes to calculate the value of *Fibonacci(N)* is much larger much sooner than for the equivalent C++ program—starting to grow at $N = 10$ while C++'s graphs starts to grow at $N = 24$. Overall, the C++ implementation is about 1170% faster than the THAL++ version. The THAL++ implementation's runtime system is a hinderance for this algorithm on a single processor machine. In theory, distributing the work evenly over multiple nodes connected via a fast ethernet switch would improve the computation time dramatically.

4.2 N-Queen Problem

The N-Queen problem is to place N queens on an $N \times N$ chessboard so that no two queens attack each other. The problem is an ideal case for a backtracking algorithm—one which if the program reaches a "dead end" on a particular algorithmic path, it *backtracks* to some previous state and begins the search with a new set of parameters. The algorithm used in this thesis is adapted from the algorithm written by Marcel van Kervinck. It also has an exponential time complexity.

Figures 4.3 and 4.4 show the code for the N-Queens program. The code used for this experiment is changed from the code in Figures 4.3 and 4.4. It doesn't allocate actors on remote nodes.

```
using printf;

class NQueenWork;

class NQueen {
  int NumProcs;
  int N;

  public NQueen(int nprocs, int n) {
    NumProcs = nprocs;
    N = n;
  }

  public int compute() {
    NQueenWork NQW;

    NQW = new NQueenWork(NumProcs);
    return NQW.compute(~(~0 << N), 0, 0);
  }
}
```

Figure 4.3 Part 1 of a THAL++ program to compute the number of solutions for placing N queens on an $N \times N$ board.

Figure 4.5 shows the results of calculating the number of solutions to the N-Queens problem for $N = 1, 2, \ldots, 12$. Again, we see that the performance time is drastically slower on a single processor than the equivalent C++ program. Overall, the C++ version is roughly 500% faster than the THAL++ version. In theory, this too should improve with more nodes connected via a fast ethernet switch.

Upon further analysis, approximately 90% of the running time for THAL++'s generated code of the Fibonacci and N-Queens programs is spent creating actors (on local nodes, in this case). The THAL runtime system does allocate pools of actors at program start-up, however the number of actors being created by these programs is quite large—numbering approximately 860000 in one case—and the pools quickly run out of preallocated actors. Future work on THAL and THAL++ should focus on optimizing memory usage and creating an actor allocation scheme that scales to large numbers of actor creations.

```
class NQueenWork {
  int NumProcs;

  public NQueenWork(int nprocs) { NumProcs = nprocs; }

  public int compute(int a, int b, int c) {
    int d, e, f;

    d = 0;
    e = a & ~b & ~c;
    f = 1;

    if (a) {
      f = 0;
      e -= d;
      d = e & -e;

      while (d) {
        NQueenWork NQW;
        NQW = new NQueenWork(NumProcs) on (a % NumProcs);
        f += NQW.compute(a - d, (b + d) * 2, (c + d) / 2);
        e -= d;
        d = e & -e;
      }
    }

    return f;
  }
}

class Main {
  public void main() {
    NQueen NQ;

    NQ = new NQueen(4, 8);
    printf(NQueen(%d) == %d\n", 8, NQ.compute());
  }
}
```

Figure 4.4 Part 2 of a THAL++ program to compute the number of solutions for placing N queens on an $N \times N$ board.

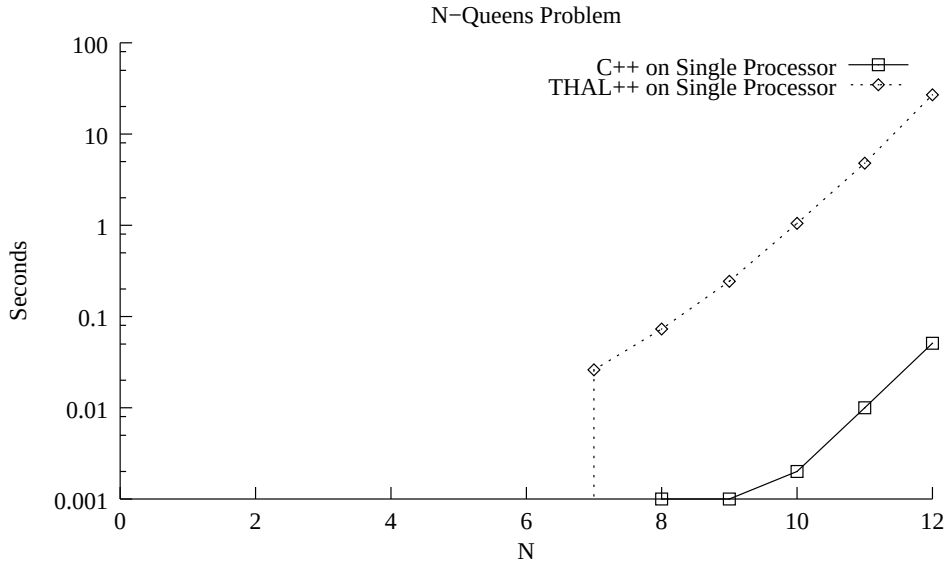

Figure 4.5 Graph of the computation time for calculating the solution to the N-Queens problem for $N = 0, 1, 2, \ldots, 12$

CHAPTER 5

CONCLUSION

This thesis presents THAL++, a new concurrent object-oriented language. THAL++ supports the actor model of computation via asynchronous message passing, concurrent call/return communication abstractions, and remote actor creation. It allows the programmer to focus on writing the program instead of the details of distributed computing. THAL++ provides features not present in the THAL language: types and a syntax that is a subset of Java.

THAL++ is compiled to C code that works with the THAL runtime system. The THAL runtime system provides an implementation of the actor model of computing. It has an efficient message passing scheme and a substrate for managing actors. Programs can be written in THAL++ which exploit the processing power of a network of computers.

REFERENCES

[1] Gul A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[2] Wooyoung Kim, *THAL: An Actor System for Efficient and Scalable Concurrent Computing*, Ph.D. thesis, University of Illinois at Urbana-Champaign, July 1997.

[3] W. C. Athas and C. L. Seitz, "Cantor User Report Version 2.0," Tech. Rep. 5232:TR:86, California Institute of Technology, Pasadena, CA, January 1987.

[4] Akinori Yonezawa, Ed., *ABCL: An Object-Oriented Concurrent System*, MIT Press, 1990.

[5] Mark Astley, *The Actor Foundry*, Open Systems Laboratory at the University of Illinois at Urbana-Champaign, February 1999, `http://yangtze.cs.uiuc.edu/foundry/`.

[6] Andrew A. Chien and William J. Dally, "Experience with Concurrent Aggregates (CA) Implementation and Programming," in *Programming Languages for Parallel Processing*, David B. Skillicorn and Domenico Talia, Eds., pp. 169–178. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[7] Carlos A. Varela, *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2001.

[8] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

APPENDIX

PROGRAMMING FOR THAL'S RUNTIME SYSTEM

THAL++ generates C code which can be read by the programmer, but it is not always obvious when inspecting the sequential C code what the variables or transformations are. Also, the relevant runtime APIs need to be documented. This is a guide for understanding these things. The bulk of the material in this appendix is from private communication with Wooyoung Kim.

A.1 Global Variables

Some global variables are used in THAL for initialization of the runtime. These must be specified. They are the following:

- `int User_thalmain_id` — Not used but needs to be defined.
- `int User_MethodIncrement` — Not used but needs to be defined.
- `int n_user_classes` — The number of classes the program has.
- `int User_AcqSizes[]` — For each class, the total size of the state variables in bytes.

- `int User_Cont_MaxSizeKey` — The largest size of the continuation objects in double words (8 bytes).

- `int User_Cont_OccurCount[]` — The number of times `new Cont_class` appears in the program. `User_Cont_OccurCount[i]` is the number of times a continuation that passes n variables and l live variables, where $n + l = i$, is called. All variables are given 8 bytes.

- `int User_MaxMsgSize` — The largest size of messages in bytes.

- `int User_Msg_OccurCount[]` — For each message size, how many times it appears in the program.

- `unsigned log2_num_classes` — The logarithm (base 2) of the number of classes in the program.

About the `User_Cont_*` Variables:

When we have a concurrent call/return communication (CC/RC), the compiler takes this call and transforms it into a non-blocking send with continuation. The computation dependent upon the replies is placed in a separate function. Once all of the values for continuing the computation are computed, the continuation is envoked. The continuation is "anonymous", so no actors know about it. The `User_Cont_*` variables contain information to facilitate memory allocation for these continuation objects.

About the `User_Msg_OccurCount` Table:

Consider a program in Figure A.1 which has three message sends. The associated `User_Msg_OccurCount` table would be `int User_Msg_OccurCount[] = { 0, 1, 1, 1 };`[1], where index 0 is for those messsages which have no parameters, index 1 is for those messages with 1 parameter, and so on. In this case, the `User_MaxMsgSize` would be 24.

```
a <- m1(1);         /* 8 bytes for args - 1 double word */
...
b <- m2(2, 3);      /* 16 bytes for args - 2 double words */
...
c <- m3(4, 5, 6);   /* 24 bytes for args - 3 double words */
```

Figure A.1 Program which has three message sends.

A.2 Global Structures and Enumerations

Structures and enumerations that are used in THAL. These hold parameters for continuations, give a unique number to classes and the public methods in them, and specify the methods and behaviors that these classes have.

- `struct Msg` — A simple structure that holds only the processor number, a pointer to the continuation, and a pointer to memory where the value from a continuation is returned.

- `struct b_`*ClassName* — A structure for the class *ClassName* that has the state variables for that class.

[1]Each parameter in THAL++ is 8 bytes.

- `enum class_enums` — An enumerated list of the classes in the program. Each entry is named `CLASS_Foo`, where *Foo* is the name of the class in the user program.

- `enum actor_enums` — An enumerated list of class methods. Each method name is unique. Each entry is named `M_Bar`, where *Bar* is the original name of the method.

- `handler User_methods[]` — An array of all of the methods in the program. The array is laid out in the following manner: if there are N classes, then the method *Quux* in class *Baz* is placed in the array at the point `(M_Quux << log2_num_classes) | CLASS_Baz`. Entries which do not have methods associated with them are given the default entry `error`.

- `struct _method_table_entry mt_`*Bar*`[]` — One of these tables exists for each class. The naming convention for these tables is `mt_Bar`, where *Bar* is the name of the class. The entries are the public methods in each class. Figure A.2 shows an example entry that declares the method *Quux* in class *Bar*.

```
{ ((M_Quux << LOG2_NUM_CLASSES) | CLASS_Bar) >> LOG2_NUM_CLASSES,
  (handler) Bar_Quux }
```

Figure A.2 Example entry into the `_method_table_entry` array.

- `struct behavior _BT[]` — An array of classes and their constructor (if any). The first entry of `_BT` should be a default *null* entry. The rest should be according to their position in `class_enum`. An entry in `_BT` looks like: `{ 0, nil, (handler)Foo_init, mt_Foo }`. If *Foo* doesn't have a constructor, then substitute `nil` for `Foo_init`.

- `struct { ... } anon_func_#_t` — A structure sent to an anonymous (continuation) function where # is a unique number given to the continuation. It holds the processor number the call originated at, a pointer to the continuation, and a pointer to memory where the value from a continuation is returned. It also has parameters which are passed into this anonymous method and any variables which are live at the point of the continuation call.

A.3 Transforming THAL++ Methods Into C Functions

The process of transforming a THAL++ method into a C function is quite involved. To maintain the namespace, methods are prepended by the name of the class.

A.3.1 Calling Conventions

Two C functions are generated for each public method—wrapper and main functions. The wrapper function parses out the parameters with which the method is called and calls the main function with those parameters. The wrapper function accepts only two parameters: the *Actor* the method is invoked upon and an *ActorMessage* structure. The *ActorMessage* structure holds information about the message including the parameters passed into that method. Those parameters are retrieved from the *ActorMessage*'s *data* field. The main function, in addition to these parameters, takes an argument for the *Actor* the method is invoked upon.

Functions for constructors take at least one argument; the *Actor* that is being initialized. If arguments are required for the constructor, then those are specified after the *Actor* argument. Inside of the constructor, the *Actor* structure's *acq* field is used as a pointer to the structure for this class which holds the internal state variables for the actor.

Functions for Private methods are generated as normal C functions.

A.3.2 Code Transformations

A.3.2.1 Actor Creation

Actor creation can be either local or remote. It is not possible during compilation to determine whether an actor creation is remote if `new`'s "`on` *Expression*" syntax is used. An actor which is created remotely doesn't have an *Actor* structure on the creating node. In that case, an *alias* is used instead. This alias acts in place of the actor on the creating node.

Figure A.3 shows the code generated for the THAL++ expression `new Foo(9.27) on` *Expression*. `mGetPN()` is a macro that returns the current node number. The number 8 in the code snippet is the size (in bytes) of the parameter 9.27. After this code, `A` is used as the actor.

A.3.2.2 Asynchronous Message Sends

Transforming an asynchronous message send involves creating an *ActorMessage* object, filling it with the parameters with which the asynchronous message send is called, locating the actor, and sending the message to that actor. Figure A.4 sends a `compute` message to

```
Actor A;
int OnNode = Expression;

if (OnNode == mGetPN()) {
  A = create_actor_locally(mGetPN(), CLASS_Foo);
  if (_BT[CLASS_Foo].initmeth)
    _BT[CLASS_Foo].initmeth(A, 9.27);
} else {
  Message MS = allocate_message(sizeof(struct message)
                      + sizeof(struct actor_create_msg) + 8);
  ActorCreationMessage ACM = (ActorCreationMessage)MS->data;

  A = malloc(sizeof(struct actor));
  *((double*)&acm->data[0]) = 9.27;
  create_actor_remote(&A->hdr.addr, mGetPN(), OnNode, CLASS_Foo,
                      8, ACM);
  A->hdr.addr = ACM->alias;
  A->hdr.vtloc = OnNode;
}
```

Figure A.3 C code for creating an actor.

actor `A` with two parameters: 37 and 9.27. Notice, when filling the `ActorMessage`'s `data` buffer, the first offset is 0, the second offset is 2, and so on.

A.3.2.3 Concurrent Call/Return Communication (CC/RC)

Transformations involving CC/RCs are done as explained in 3.3.3. CC/RCs do change the public functions in a few ways, however. A public function which is the target of a CC/RC is modified to accept a pointer to the *Msg* structure as an extra parameter. The *Msg* structure holds the process number, a pointer to a continuation object, and a pointer to memory where the result of the computation is to be stored.

The `fAllocContinuation(2)` function call allocates a *Continuation* object that passes along with it two values (parameters and/or live variables). The `synch_counter` vari-

```
ActorMessage AMsg = (ActorMessage)alloc_actorMsg(2 * sizeof(double)
                                    + sizeof(struct actor_message));
int msg_id;

if (A->hdr.vtloc == mGetPN()) {
  AMsg->receiver = A;
} else {
  AMsg->receiver = lookup_actortable(A->hdr.addr.birthplace,
                                    A->hdr.addr.address)->definition;
}

*((long*)&AMsg->data[0]) = 37;
*((double*)&AMsg->data[2]) = 9.27;
msg_id = M_compute << log2_num_classes;
send_simple(A->hdr.addr.birthplace, A->hdr.addr.address, msg_id,
            16 + sizeof(struct actor_message), AMsg);
```

Figure A.4 C code for sending an asynchronous message to an actor.

able indicates that one computation needs to be performed before the continuation can be called. The `callback` variable is the anonymous continuation function to call once the `synch_counter` becomes zero. The `env` variable holds the processor number, a pointer to the *Continuation* object, a pointer to the point in memory to place the computed value for the anonymous function, and any parameters and live variables.

Figure A.6 shows some code for returning a value from a function that is the target of a CC/RC. The return value in `VALUE` has been computed and needs to be sent to the anonymous continuation function. If the actor is local, then we can assign the value directly otherwise we need to send it to the correct actor. Once the synchronization counter (`synch_counter`) becomes zero, the result is sent to the anonymous continuation function.

```
void Foo_compute(Actor this, int i, Msg* M) {
  int pn = M->pn;
  Continuation cp = M->cp;
  void *mp = M->mp;
  struct b_Foo *acq = (struct b_Foo*)this->acq;
  Continuation C_anon_func_1;
  int LV;

  /* Some code */

  C_anon_func_1 = fAllocContinuation(2);
  C_anon_func_1->synch_counter = 1;
  C_anon_func_1->callback = (ContMethod) anon_func_1;
  ((anon_func_1_t *)C_anon_func_1->env)->ProcessorNumber = pn;
  ((anon_func_1_t *)C_anon_func_1->env)->Cont = cp;
  ((anon_func_1_t *)C_anon_func_1->env)->Memory = mp;
  ((anon_func_1_t *)C_anon_func_1->env)->lv0 = LV;

  /* Asynchronous message that uses C_anon_func_1 as the continuation */

}
```

Figure A.5 C code for filling a continuation.

```
if (pn == mGetPN()) {
  /* This actor is local. Just assign... */
  *(long*)mp = VALUE;
  if (--cp->synch_counter == 0)
    cp->callback(cp, NULL);
} else if (pn != -1) {
  send_reply(pn, cp, mp, 0, (void*)VALUE);
}
```

Figure A.6 C code for returning from a function that is the result of a CC/RC call.

VITA

William Clarence Wendling, Jr. was born in Springfield, Illinois. He graduated from the University of Illinois at Urbana-Champaign with a B.S. in Mathematics. So, of course, he became a computer programmer. After two less than satisfying jobs as a programmer in the Real World, he took a job at the National Center for Supercomputing Applications. The building in which he first worked at NCSA was the birthplace of Mosaic and was at one point condemned. He is currently working with the Hierarchical Data Formats (HDF) group at NCSA.