

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

Tristan Aubrey-Jones

May 2008

Investigation into alternative programming
abstractions using “Causal systems”

Project supervisor: Prof. Michael Butler
Second examiner: Dr. Bernd Fischer

A project report submitted for the award of
Computer Science MEng

Abstract

A project investigating and developing an implicitly concurrent programming language, based on a metaphor taken from the physical world is reported. The project is introduced and key background ideas explained making the case for a programming paradigm where programs consist of systems of autonomous agents, or active objects which communicate via message passing. A literature search researching the development of a similar paradigm called the “Actor model” is reported and criticised. A language enhancing Java with actors and linear types is defined and key decisions are discussed. Translation rules to reduce the language into Java are presented, and a prototype translator is developed. Example programs are written, compiled, and executed to evaluate the usefulness of the language. Conclusions are drawn and the language found to provide a familiar notation for implicit parallelism, and a compelling new model for concurrency, combining the performance of shared variables with the elegance of message passing. Finally further work is suggested to extend and refine the language, and its implementation.

Contents

<i>Abstract</i>	2
Contents	3
Acknowledgements	5
1 Introduction	6
1.1 Background.....	6
2 Literature Search	9
2.1 The History of the Actor Model	9
2.2 Model Evaluation.....	10
3 Language Definition	11
3.1 The Actor Class	11
3.2 Inheritance and Interfaces	13
3.3 Resource sharing.....	14
3.4 Reaction statements	15
3.4.1 Concurrency within Reactions	15
3.4.2 Control flow within Reactions	15
3.4.3 Statement Grammar.....	16
3.5 Extended Language Definition.....	16
3.5.1 Expression Actors	17
3.5.2 Fork blocks	17
3.5.3 Sequential Expression Actor Invocations.....	18
3.5.4 Actor events	18
3.5.5 Condition Reactions	18
3.5.6 Message Declaration Shorthand.....	19
3.5.7 Extension Grammar.....	19
4 Language Translation	21
4.1 The Minimal language	21
4.2 The Extended language	22
5 Translator Implementation.....	26
5.1 Translator Design.....	26
5.2 Runtime Library.....	28
5.3 Translator Implementation	28
5.4 Testing.....	29
6 Evaluation and Findings	30
6.1 Dining Philosophers.....	30
6.2 Parallel Quicksort	31
6.3 Calculator App.....	32
6.4 Conclusion.....	33
6.5 Further Work	34
7 References	36
Appendix A.....	39
Initial Syntax Experiment	39
Appendix B.....	44
Resource sharing experiments.....	44
Resources as Actors and Linear Types.....	48
Appendix C.....	49
Minimal Language Grammar.....	49
Extended Language Grammar.....	51

Appendix D.....	54
Minimal Language Translation Rules	54
Extension Translation Rules	57
Appendix E.	67
Actor Base Class	67
Translator Test Program	70
Appendix F.	73
Dining Philosophers Example.....	73
Quick sort example.....	75
Event actor class.....	77
Calculator example.....	78

Acknowledgements

I would like to thank my supervisor Michael Butler for his invaluable guidance and discussion, and for first suggesting some sort of ownership passing as a solution to the resource sharing problem. I would also like to thank Gul Agha for his foundational work on the actor model, which provided a formal underpinning for the idea.

I thank my family, and housemates for their constant love and affection, and most of all I thank my Lord and God Jesus the Christ for his unceasing river of love, and eternally magnificent nature which formed the basis of this idea and from whom, through whom and to whom are all things. To him be the glory for ever! Amen¹.

¹ Romans 11:36

1 Introduction

As programming languages have progressed they have been abstracted further away from describing a sequence of machine dependant operations, and have favoured modelling an abstract algorithm which can then be made machine specific by compilation or interpretation. However, the implicit assumption that the algorithm designed is going to be realised on a *single* Turing machine has stayed engrained in almost all notations.

This is a false assumption as there are many occasions where steps could be reordered, or performed concurrently without adversely affecting the result. Furthermore with the advent of multi-core, multi-processor and distributed architectures, it is now commonplace for machines to support at least some measure of parallel execution². If we want to write programs that effectively make use of these machines, we must forget this assumption and reform our programming notations accordingly.

The goal of the project is to investigate, and start development of a language for concurrent programming, based on a metaphor of concurrency taken from causation and the physical world, aimed at making better use of modern multi-processor computers. This will be achieved by building on the Actor model of computation, to design a language with similar grammar to an object oriented language so that it would be easy for an object oriented programmer to learn. Possible language constructs will be investigated and chosen to enable powerful yet understandable modular programming. A further aim is to produce a language grammar, and implement a translator for the language that will output Java code which could be executed by multiple processors on a single machine.

1.1 Background

If we liken a program to a play in the theatre, then traditionally there would be a single processing unit that would play all the parts, turn by turn. Now we have machines where we have a cast of processing units, all available to play different parts. If we are to make proper use of these machines we need to stop writing the scripts specifically for one actor, but rather for a full cast. In this way the play can be performed in different ways depending on the size of the available cast. The same program would still be executable by a single processor, but it could also be efficiently partitioned over any number.

This is already possible to some extent as most modern operating systems provide facilities so that several separate processes can execute simultaneously. The limitation of the current approach is that for concurrent execution to be possible the processes must be separate, each programmed as a distinct single sequence. This is acceptable when a problem is embarrassingly parallel³, so that it can be easily divided into independent parts, but many problems are highly interdependent “or fine grained” [28, 33] and cannot be so easily separated. It is possible on some platforms to have multiple “threads” of execution with shared memory, but each thread is still a single sequence and so

² E.g. As of May 2008 over 36% of the user’s of the Steam online gaming network use 2 CPUs [27]; Intel revamps its road map to focus on multi-core architectures [29].

³ Such as Monte Carlo calculations [32].

nontrivial interactions between threads require complex synchronization and are very difficult to design and debug.

A far better approach would be to incorporate flexible parallelism into our fundamental programming constructs [28], by introducing a measure of ambiguity in our programming model. In traditional imperative programming each statement can only be executed, after the previous one. This constraint is critically important in some places, without which the program behaviour changes, but in other places it may be an unnecessary stipulation. By making this constraint universal we limit the kind of devices that we may use to execute the code, and especially make efficient concurrent execution very difficult [25].

The sequential constraint has its roots in one of the most fundamental of all physical laws: “causation”, where each action causes further actions, which will cause others, forming chains of temporal dependency. A single threaded program can be thought of as a single causal chain where each instruction is *caused* by the completion of the previous one. However this is only a special case of the real world scenario, when in reality many actions occur at once, each of which being caused by a subset of the actions that preceded it.

To introduce the parallelism we see in the real world into our programming we need to somehow specify an action’s causal dependencies, so that when order matters sequence is forced but mutually independent actions may be executed in any order, even concurrently. In this way we not only support instruction level parallelism and task parallelism, but the parallel execution of sections of code throughout our programming. This is also highly desirable as it means that when programming we do not specify much more than the minimum causal constraints required for the algorithm to perform correctly, giving the potential for the maximum possible concurrency.

In the physical world actions don’t just happen abstractly in the ether, they are associated with objects, and affect objects. If a non existent tree falls down in the forest, and doesn’t affect any real object, then it doesn’t really fall. Correspondingly every “effect” that has been caused, must have affected some real object. So for any parallel programming model to be useful, it must have some notion of separate “objects”, or areas of memory, whose state can change in consequence to an action. Thus the clear separation and grouping of independent state variables, is key to enabling concurrency, as if there is only a single conceptual “state” only one action can affect that state at a time forcing linear execution.

Applying these basic conclusions drawn from how the world works provides a natural way to program, where instead of issuing a sequence of commands that a single machine must obey, we describe an abstract environment (or causal system) of distinct, but interacting objects, which can be simulated on virtually any specific computational device. The objects could be distributed over many individual nodes interacting via a communication network, and mutually independent effects could be evaluated by separate processing units within each node.

This is similar to the object oriented programming paradigm, only instead of passing a single thread of control flow between object methods, notifications are received by objects causing reactions which potentially affect the object’s state and send further notifications to other objects. In this way each unique object has its own independent

state, and a mapping between notifications and reactions, or causes and their effects. This is like designing a machine with separate interacting components so we do not define a precise unalterable algorithm, but a model or machine with causal properties, thus giving a clear modular structure and allowing intuitive concurrent programming. This model provides a natural way to think about parallel computing and it is this model of programming that will be investigated and developed further.

2 Literature Search

This model of computation is almost identical to one revealed in the literature search, which identifies Hewitt, Bishop and Steiger as the first people to suggest it, calling it the “Actor model of Computation” [1]. This is very similar to the idea suggested earlier where the objects in our environment are called “actors”, which interact with each other via message passing [2].

Here a program describes an “actor system”, which is almost identical to the environments or “causal systems” described earlier, where each actor may receive messages and respond to them by changing their internal state, creating more actors, and sending further messages. Thus actors are active objects which may act concurrently, reacting to causes when messages are received, by affecting effects on themselves, and causing further effects by sending further messages. This allows sequences of execution (or causal chains) to be achieved by each step sending a message, to trigger the next.

2.1 The History of the Actor Model

The Actor model of computation was first proposed in 1973 by Hewitt, Bishop and Steiger at the International Joint Conference on Artificial Intelligence [1]. It was proposed as a universal formalism for artificial intelligence, and the first operational model was published by Irene Greif in 1975 for her doctoral dissertation [2]. Two years later Hewitt and Baker published the fundamental laws for actor based computation [3], and a paper demonstrating the model’s ability to describe existing control structures [4]. Clinger developed the first denotational semantics for his doctoral dissertation in 1981 [5], by using Plotkin’s theory of “power domains” [6]. In 1985 a paper introducing Act3, an actor language based on extending a group of Lisp machines, was published [7], and the next year Agha published a paper describing SAL (Simple Actor Language) [8] which incorporated a number of language extensions, most notably “continuations” to allow reactions to wait for responses from other actors before completing.

By this point the model itself had been clearly formalised, and over the next 15 years research was done by Agha and others with a view to develop an actor based language for open distributed systems [9 - 21]. This included work focussed on developing mechanisms for large scale distribution [10, 11, 16], and techniques for modularisation and abstraction in actor languages [15, 17, 19]. The papers published in these years also aimed to advocate the model to the wider research community. This research culminated in the development of an actor based language for open distributed systems called SALSA (Simple Actor Language System and Architecture) [21] in 2001.

In the past 7 years the SALSA compiler has been developed⁴, and meanwhile at Berkley ideas from the actor model have been used in the development of “actor-oriented programming” [22 - 25]. This takes the fundamental concepts of the actor model, but makes an actor’s internal implementation heterogeneous. Actors have well defined interfaces and can be inherited to allow a clear modularity [24]. When programming, actors can be visually interconnected such that writing a program is very reminiscent of modular electronic design [23].

⁴ See <http://wcl.cs.rpi.edu/salsa/>

Despite the progress in this field, there still does not exist a purely actor based language for general purpose programming, and specifically for deployment on multiprocessor architectures.

2.2 Model Evaluation

The Actor model as described by Agha [8] is a powerful model for computation that very much follows the physical metaphor described earlier. A huge advantage is that concurrency is one of its fundamental tenets, rather than an inconvenient add-on. Furthermore the use of message passing as the only mechanism of inter-agent communication means that concurrent access to a single resource or area of memory is implicitly prevented.

However this blessing can also be a curse. A downfall of the actor model is that “any object passed as an argument is cloned at the moment it is sent, and the cloned object is then sent to the target actor” [26]. This copying could be unnecessary, particularly when implementation is on a single, multiprocessor system. Furthermore if a resource needs to be shared and not copied, it must be implemented as an actor, and accessed via message passing (see Appendix B for relevant examples). This may sometimes be convenient, but there are often cases where a resource may need to be exclusively accessed by one actor at a time, and where direct access to the resource is desirable as the message passing overhead may be unacceptable.

The language designed here will therefore follow the Actor model for the most part, taking advantage of prior research into its semantics, whilst at the same time solutions to its deficiencies are proposed and employed.

3 Language Definition

Prior to the discovery of previous work on the Actor model, experimental programs were written using an almost identical model. For example a graphical pocket calculator application was written and is listed in appendix A. In these experimental notations, objects directly mimicked objects in the real world. Every existent thing has 3 facets: the actuality of its being (the thing in itself), an externally visible image, and the ability to interact with other objects⁵. In computer science this universal principle is expressed in the model-view-controller design pattern, where the model stores an entity's state, the view projects an external image of that state, and the controller allows iteration with it. In these early experiments each object had state variables, would be visible through "observers", and would interact with others by reacting to messages.



After considering various abstract syntaxes based on regular expressions and extending state machines to include predicates, it was decided that an approach borrowing from object oriented class declarations would be more suitable. Later prototypes featured "reactions" like method calls, each of which defined how to react to a certain message type.

Since the discovery of existing research on the Actor model these ideas have been refined and extended, resulting in the language definition that follows. Much of the research on the Actor model has focussed on functional notations, thus alienating it from those more comfortable with its closest natural neighbour, the object oriented programming paradigm. Thus the language presented here uses a Java-like syntax, as an extension to Java, to minimise the learning curve for those familiar with similar languages. This chapter gives an overview of the syntactic design of the language, and discusses some of its key design decisions.

3.1 The Actor Class

The kernel of an actor language is the grammar for declaring an actor behaviour type. A behaviour type defines a causal mapping between incoming messages and tuples of: new behaviours to adopt, new actors to create, and messages to send. The first actor languages defined behaviour types which contained state parameters and a function that was evaluated when a message was received [8, 9]. This function branched on the message's type and actor's state, and contained "replacement behaviour" expressions to specify the behaviour type and state that the actor should adopt when responding to a subsequent message. Messages were received in the context of the destination actor's current behaviour, executing its behaviour function and defining a new behaviour in which any following message is received. Thus an actor's lifetime consisted of a chain

⁵ Idea derived from the Christian doctrine of God as trinity, one God, yet in three persons: Father (the actuality of his being), Son/Word (the revelation or exact representation of his being) and Spirit (the influence and presence of his being that interacts with creation). See <http://en.wikipedia.org/wiki/Trinity>.

of behaviour states, where incoming messages cause the actor to pass from one state to another.

This system of switching behaviour though conceptually elegant is not intuitive. Furthermore letting actors change their behaviour type during runtime, requires a dynamic type system counter to Java's static typed class system. Thus a system has been designed that is theoretically equivalent, using an "actor class" construct to declare behaviour types with state variables that persist across message deliveries, such that actors can vary their behaviour by modifying these state variables and then branching on them, whilst their type remains constant. These actor classes must define how an actor should react when it receives a message, in order to specify the causal mapping, between an incoming message and the effect that it causes.

The `switch` and `if` statements in Java can be adopted without modification to allow selection based on the actor's state and the value and type of the message. The grammar for an actor class definition could either define a single block of `if` statements, or a number of separate members. SAL actor behaviours define a single block, but its top level is commonly a series of `if` statements branching on the incoming message type [8] but in Java this would require excessive typecasting. SALSA defines named "message handlers" to which messages are sent, decomposing the block based on an implicit "message handler name" parameter in every message [26]. These handlers look very similar to methods and so would be familiar, but it may not be distinct enough, so that the concept of receiving messages could be confused with method invocation. Furthermore this syntax requires the handler desired to be explicitly coded rather than being selected by value, preventing the possibility of automatically invoking a different reaction based on some value. Allowing messages to be arbitrary objects allows them to be actors, and encourages actors to be designed as agents reacting to stimuli rather than collections of related subroutines. For the added flexibility it provides, and its more natural modelling of actor behaviour, the language uses anonymous reactor members, which simply define the message object type that they handle.

An example actor class definition and illustration of its meaning follow:

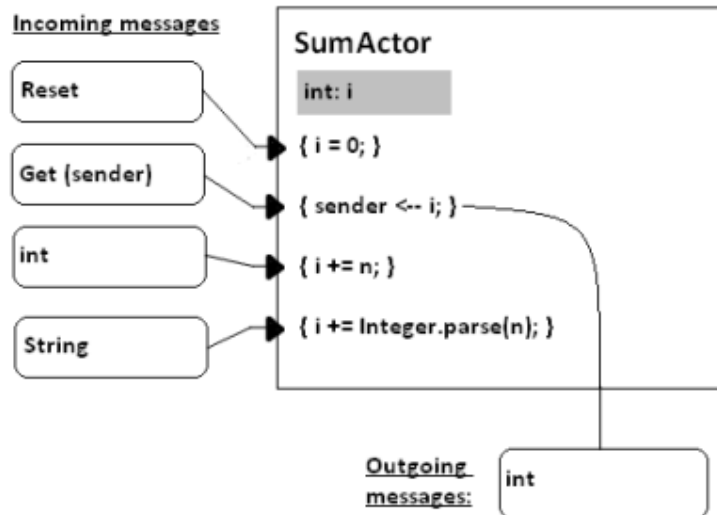
```
public aclass SumActor {
    int i = 0;

    class Reset {}
    react (Reset r) { i=0; }

    class Get { Actor sender; }
    react (Get g) { g.sender <-- i; }

    react (int n) { i += n; }

    react (String n) { i += Integer.parse(n); }
}
```



[Figure 1: Diagram illustrating the SumActor actor class example.]

3.2 Inheritance and Interfaces

An advantage of the decision to define an actor's behaviour using a number of reactor members is that it makes actor class inheritance possible, and allows the definition of actor interfaces. An actor class that "extends" another inherits all of the fields and reactor members in the base class. This facilitates code re-use and allows the construction of progressively more complex actors. Further to this, because messages are statically typed it is possible to define actor interfaces, that when implemented guarantee that an actor defines a reactor for the message types defined.

The grammar for defining actor classes and actor interfaces is defined below. Only the major productions are defined here; for the full language definition refer to Appendix C.

```

type_declaration      ::= modifiers ( actor_class | actor_interface
                                | java_class | java_interface )

actor_class            ::= ( aclass | actor ) identifier
                        [ extends data_type ]
                        [ implements data_type_list ]
                        actor_class_body

actor_class_body      ::= "{ " { modifiers actor_class_member } "}"

actor_class_member    ::= react "(" data_type identifier ")"
                        statement_block |

                        type_declaration |

                        data_type
                        field_declarator { "," field_declarator }
                        ";" |

                        identifier "(" expression_list ")"
                        constructor_block |

                        data_type identifier
                        "(" parameter_list ")" statement_block
  
```

```

actor_interface      ::=  ainterface identifier
                        [ extends data_type_list ]
                        actor_interface_body

actor_interface_body ::=  "{ " { modifiers actor_interface_member }
                        "}"

actor_interface_member ::= react "(" data_type identifier ")" ";" |
                           type_declaration |
                           data_type
                           field_declarator { ",", field_declarator }
                           ";" |

```

[Figure 2: BNF for actor classes and actor interfaces in minimal language]

3.3 Resource sharing

As was discussed previously, a severe limitation of the actor model is its failure to adequately support the sharing of resources between actors. Messages are sent by-value, and actors by-reference and so shared resources must be implemented as actors. This is not always intuitive and is often inefficient, as a resource may be a large data structure where duplication is expensive, and where direct access to the resource is desirable. A further problem arises if a resource actor needs to be locked so it can only be used by one actor at a time. This behaviour can be achieved by using a secretary actor to buffer incoming messages whilst it is “locked”, but it is far from elegant. Appendix B lists a queue actor which can be locked, such that it blocks other requests until unlocked, but the code is convoluted and prone to the kind of bugs that existing constructs for concurrency suffer from.

There seem to be a number of circumstances where it would be useful to be able to guarantee that only one actor can have access to some object or actor, but still be able to share this exclusive access between actors. Given that the actor model already defines a construct for the transference of a message from one actor to another, one possible solution to this problem is to extend this to allow the transference of resource ownership between actors. Messages could either be copied when sent, or transferred such that the sending actor can no longer access the resource once it has been transferred. The complication here becomes how to police the ownership, so that once transferred a resource cannot be accessed by the sender.

A method considered to achieve this was to define special resource fields within actors, which can be populated when a resource is received and then destroyed when transferred. However forcing the resource to reside in a special field means that it could not be passed to a function and could only be transferred into equivalent fields. A better method defines actor and object “resource types” which may never be referenced by multiple actors, so that resources can be received like normal messages.

The solution employed builds on the two discussed above to define certain types that can only be referenced by one identifier at a time, so that every read is a destructive read. These types cannot be assigned, but only transferred between identifiers and between actors so each object is guaranteed to only be accessible through one identifier in one actor at any one time, and yet the reference can be passed between actors. This avoids

introducing further complexity as the message passing metaphor is already essential and efficiency is improved as many messages can be transferred by reference, enabling significant optimizations on shared memory architectures. Prior work on just such a method of typing refers to these as “linear types” [30, 31]. Experiments comparing the use of such types, with making resources actors and using mutual exclusion locks can be found in Appendix B. This approach gives the efficiency of variable sharing IPC with the understandability of message passing, eliminating the need for complex synchronization techniques and the subtle bugs their misuse can cause.

3.4 Reaction statements

Decisions have to be made regarding the nature of the code in reactor blocks. Early notations were extensions to functional programming languages so there was no mandatory evaluation order for expressions. Apart from the precedence of operands over operators, expressions could be evaluated concurrently. The question is therefore, should similar parallelism within reactions be assumed in this language, or should statements execute sequentially?

3.4.1 Concurrency within Reactions

The first actor languages could specify ambiguity in execution order because behaviour’s state parameters were never mutated once defined, so reactions did not mutate shared variables but defined replacement behaviours using `become` expressions. Imperative programming however, necessitates linear execution order for statements because statements modify global state rather than compute functions of operands. Thus statement execution in reactor blocks must be fundamentally sequential although the inclusion of a construct to allow the parallel execution of some statements is possible.

The simplest construct to achieve this is the `par` block [34]. This is a composite of statements, all of which may be executed in parallel. The major limitation of such a construct is that to be of use its children must be able to modify shared variables defined in higher scopes. This would necessitate an additional construct for variable locking thus blurring IPC techniques. The possible restriction of giving `par` blocks read only access to variables in higher scopes has been investigated, but shown to severely diminish their usefulness. Furthermore equivalent parallelism can already be defined by creating new actors. Therefore as support for block-oriented parallelism adds nothing to the power of the language and creates problems with variable sharing, it is not implemented as a fundamental construct.

3.4.2 Control flow within Reactions

If reaction statements are to be sequential the next question is what control flow constructs should be incorporated? Java supports method invocations, loops and the `goto` statement, but these may not be needed or appropriate. As was discussed previously, pure actor languages do not define any sequence so an actor’s behaviour can be thought of as an instantaneous “reaction”. Control flow is specified solely through message passing. Similarly any desired computation can be described using a minimal language where reactor blocks are simple sequences of statements (including conditional branches) so no further control flow is required. Iteration can be performed using tail-recursion, and general recursion through the creation of new actors. Method invocations

can be achieved by sending a request message to an actor, and then reacting to a response message returned to the caller. If the method call should be performed synchronously incoming messages to the caller actor should be buffered until the response is received. A syntactic sugar for just such a technique is developed later in section 3.5.

Thus block-oriented iteration and method calls are not required, in fact their use deviates from the strict Actor model, as reactions may not ever terminate. The decision to include them must be based on whether their convenience warrants this deviation. If Java's large existing class library is to be leveraged method invocation must be possible, despite the fact that this could lead to very computationally expensive or eternal reactions. And since this is required, "internal methods" whose blocks have the same syntax as reactor blocks have been incorporated into the actor class's grammar. These increase code reuse within an actor's definition, and can be used for recursion and thus finite iteration via tail recursion. Since tail recursion is possible the addition of loop constructs would not add any extra possible complexity; however the decision to omit them has been made to encourage the implementation of short "instantaneous" reactions that make full use of message passing discouraging long winded reactions.

3.4.3 Statement Grammar

Important productions for defining a reactor statement block in the minimal language is defined in Backus-Naur form as follows. Productions are omitted; for the full language definition refer to Appendix C.

```
statement_block      ::=  "{" { block_statement } "}"
statement            ::=  ";" |
                           statement_block |
                           if_statement |
                           switch_statement |
                           return_statement |
                           break_statement |
                           expression_statement
expression_statement ::=  expression ";"
expression           ::=  expression1 [ assignment_op expression1 ]
assignment_op        ::=  "=" | <-- | += | -= | etc...
```

[Figure 3: BNF for reactor statements]

3.5 Extended Language Definition

The language definition presented up until this point provides a minimal actor language that implements the whole Actor model and so is sufficient to express any construct required [4]. This section describes some additional syntactic sugars for common programming patterns, all of which can be translated into the minimal language.

3.5.1 Expression Actors

One very common programming pattern is the request response pattern where some action is requested and once complete should return some information to the requester. This pattern can be explicitly written in terms of message sending and actor creation, but it is long winded and so a syntactic sugar for it has been devised. The shorthand is based on the concept of an “expression actor” as described by Agha [8] where messages sent to an expression actor contain a reference to the actor to send the result to.

```
aclass ExpressionActor extends Actor returns int {  
  class Add { int a; int b; }  
  react (Add add) {  
    return add.a + add.b;  
  }  
}
```

This shorthand allows some actor classes to define a return type, which must be returned by all of its reactions as above.

3.5.2 Fork blocks

For expression actors to be of use a simple means of invoking them and handling their response messages is desirable which does not mandate the explicit declaration of reactors to handle responses. Furthermore requests often need to be synchronous, as the calling actor may need the result of the invocation in order to complete its reaction.

A shorthand providing a simple mechanism to achieve this has been designed called the **fork** block. This construct is much like a **let** expression in Lisp where a number of expression actors may be invoked and their results assigned to local variables within a block, which is executed when they have all responded.

```
1: ExpressionActor expActor = new ExpressionActor();  
2: fork ( int sum1 = expActor(new Add(1, 2));  
3:       int sum2 = expActor(new Add(5, 6)) )  
4: {  
5:   this.value = sum1 * sum2;  
6:   System.out.println("Fork complete");  
7: }  
8: System.out.println("Fork called");
```

Here an instance of an expression actor class is invoked using a similar syntax to a method invocation expression. When the **fork** block begins, requests are sent to all of the callee actors and when all the replies are received the continuation block executes. Once the requests have been sent execution continues sequentially, so that the forks are like asynchronous method calls and their bodies are like call-back functions.

As this construct creates the possibility that multiple fork bodies execute concurrently it must be semantically verified that only one of any possibly multiple concurrent forks access variables in a higher scope. This restriction isn't too onerous as forks are often nested so they execute sequentially (see next section) and even when parallel they often only require access to the results of their invocations.

3.5.3 Sequential Expression Actor Invocations

Operations are frequently required to be executed in sequence and so a further shorthand has been incorporated to allow expression actors to be invoked sequentially like the left hand example below:

<pre>int a = expActor1(1); expActor2(a * 3); int b = expActor3(a); fork(e4(1); e5(2)); System.out.println(b);</pre>	<pre>fork (int a = expActor1(1)) { fork (expActor2(a * 3)) { fork (int b = expActor3(a)) { fork (e4(1); e5(2)) { System.out.println(b); } } } }</pre>
---	---

Here expression actors are invoked in sequence such that each is sent its request only once the previous has responded. This translates into nested fork blocks as per the right hand example above.

3.5.4 Actor events

A further design pattern that is particularly useful is the publish-subscribe or observer pattern used in event-based programming [34]. Objects define “events” which may be triggered, and to which multiple “event handlers” subscribe such that they receive notifications when the event is triggered. The message sending primitive in actor programming makes the language ideally suited to event-based programming, as event handlers can be triggered asynchronously and event handlers execute concurrently. No extra syntax is required for the definition of event handlers, as actors themselves are aptly suitable, but a useful shorthand to create publicly accessible events has been defined to provide a concise method for subscription.

3.5.5 Condition Reactions

Another extension that has been investigated is the ability to react to conditions on local state variables. This construct was first considered in initial experiments and has been found to provide a simple mechanism for iteration, and a useful mechanism to hook complex state changes.

```
aclass Counter {
    int i = 0;
    class Inc {}
    react (Inc m) { i++; }
    react-when (i % 10 == 0) {
        System.out.println(i);
    }
}
```

The example above is a counter which displays the current count every 10 increments.

3.5.6 Message Declaration Shorthand

In order to avoid repetitive definition of message object types, a syntactic sugar has been defined to produce a passive object class from a named parameter list, as follows:

```
message Message1 (int n, String s);
```

3.5.7 Extension Grammar

Modified and additional productions for the extended language are given below. For the full extended language definition, please refer to Appendix C.

```
type_declaration      ::= modifiers ( actor_class | actor_interface |
                                   message_type | java_class | java_interface )

actor_class            ::= ( aclass | actor ) identifier
                        [ extends data_type ]
                        [ implements data_type_list ]
                        [ returns data_type ]
                        actor_class_body

message_type           ::= message identifier "(" parameter_list ")";

statement             ::= ";" |
                        statement_block |
                        if_statement |
                        switch_statement |
                        return_statement |
                        break_statement |
                        expression_statement |
                        fork_statement

fork_statement         ::= fork "(" { fork_exp } ")"
                        [ statement_block | ";" ] |
                        fork statement_block

fork_exp               ::= local_variable_declaration |
                        expression_statement

actor_class_member     ::= react "(" data_type identifier ")"
                        statement_block |
                        react-when "(" expression ")"
                        statement_block |
                        type_declaration |
                        data_type
                        field_declarator { "," field_declarator }
                        ";" |
                        identifier "(" expression_list ")"
```

```

        constructor_block |

        data_type identifier
        "(" parameter_list ")" statement_block |

        event type_name identifier
        { ",", identifier } ";"

actor_interface_member ::= react "(" data_type identifier ")" ";" |

        type_declaration |

        data_type
        field_declarator { ",", field_declarator }
        ";" |

        event type_name identifier
        { ",", identifier } ";"

```

[Figure 4: BNF for extended language]

4 Language Translation

The language defined previously extends Java and can thus be implemented by translating the extensions into plain Java. This chapter presents and explains some key translation rules. These translation rules could be modified to optimize the performance of the java emitted, but those given lead to correct execution and are used in the prototype compiler. The extended language can be expressed in terms of the minimal language, and so the rules are organised into a separate section for each. For the full list of translation rules please refer to Appendix D.

4.1 The Minimal language

The essential element of the language is the actor class, which can be realised as a java object with its own thread, so actor classes translate into java classes which inherit from the common `ajava.lang.Actor` class (see Appendix E). Thus actor inheritance is implemented through class inheritance, and actor interfaces can be enforced using java interfaces. Each member of an actor class translates into one or more java class members. Fields and private methods are simply subsets of the java syntax and so need little modification. Each reactor translates into a publicly accessible ‘`deliver`’ method which buffers the incoming message, and a private ‘`react`’ method which contains the actual reaction behaviour.

```
f3(reactor_member(message_type, message_id, reactor_number, block))
=
    public void deliver(message_type message_id) {
        bufferMessage(new ActorMessage(message_id, reactor_number));
    }

    protected void react(message_type message_id)
        f5(block)
```

A deliver method takes one parameter, the message send, and appends it to the actor’s message queue along with a reactor number identifying its corresponding ‘`react`’ method. Every actor class overrides the protected ‘`processMessage`’ method, which dequeues the message, casts it to the correct type based on its reactor number, and invokes its react method.

The final key element of the minimal language is the transference operator which sends messages to actors and transfers linear objects between identifiers.

```
f7(transfer_expression(lhs, rhs))
    If lhs is an actor.
=
    If rhs is a basic type OR rhs is a class creation expression:
        lhs.deliver(rhs);

    Else
        lhs.deliver(rhs.clone());

    Else
=
    lhs = rhs;
```

Transference expressions take the form ‘lhs <-- rhs’ and translate into the invocation of a ‘deliver’ method if the destination is an actor or an assignment if it is a passive object. If the ‘rhs’ expression is of a linear (or basic) type it is passed as is, by reference, otherwise the expression translates to invoke the ‘clone()’ method on the original expression. Finally to make linear object reads destructive any statement that contains a reference to a linear type translates to the same statement and a null assignment to that reference.

4.2 The Extended language

Expression Actors

Expression actors define a return type, so that they can be invoked like methods, and so reactions return values when they complete. To do this incoming message types must include a reference to the calling actor so return statements can be translated into message sends to this actor. Thus a response message class is generated for the actor and a nested request class is generated for each reactor that encapsulates its message type and the calling actor.

```
translate_exp_member(reactor_member(message_type, message_id,
                                   reactor_number, block), actor_body)
=
private static class message_typeRequestMessage
    extends ajava.runtime.ActorRequestMessage
{
    public message_type value;
    public message_typeRequestMessage(final Actor rsvp, final int reqId)
    {
        super(rsvp, reqId);
    }
}

react (message_typeRequestMessage reqMessage) {
    message_type message_id = reqMessage.value;
    translate_reactor_block(block)
    create_condition_checks(actor_body)
}

declare_fork_actors(reactor_member)
```

Every reactor translates into one which receives the corresponding request message class, and an initial statement is injected which defines a local variable initialized to the request’s payload so the request message itself is directly accessible by return statements.

```
translate_stmt(return_statement(return_value))
=
{
    Response responseMessage = new Response(requestMessage);
    responseMessage.value = return_value;
    requestMessage.sendReply(responseMessage);
    return;
}
```

Finally every return statement translates as above, sending a response message to the requesting actor.

Fork blocks

The fork block is a powerful extension to the language, allowing multiple expression actors to be invoked concurrently, and a “continuation” block to execute when all of the invocations have completed. They are realised using delegate actors which wait for responses, executing their bodies when all the responses have been received. Thus every fork primarily translates into an actor class, with its local variables translated into field members, and with reactors to receive response messages. These classes are nested within one another in the same hierarchy as the original reaction, so that a fork defined within another fork can access and modify variables in higher scopes. A primary delegate actor is generated for every reaction that contains forks, to contain the message field and to send the actor a message when its forks have completed, preventing further reactions executing until it has done so.

```
declare_fork_actors(fork_statement(fork_id,
                                   statement_1, ..., statement_n, block), container_name)
=
  aclass fork_idForkActor {
    container_name OWNER_RECEPTIONIST
    int MSG_WAITING_COUNT;
    int FORK_WAITING_COUNT;

    declare_local_vars(block)
    declare_local_vars(statement_1)
    ...
    declare_local_vars(statement_n)

    public fork_idForkActor(final container_name OWNER_RECEPTIONIST)
    {
      this.OWNER_RECEPTIONIST = OWNER_RECEPTIONIST;
      this.MSG_WAITING_COUNT = count(statement_1, ..., statement_n);
      this.FORK_WAITING_COUNT = count_forks(block)+1;
    }

    For every unique type of expression actor referenced in statement_1,..., statement_n:
    declare_fork_reactor(expactor_type_1, statement, ..., statement)
    ...
    declare_fork_reactor(expactor_type_n, statement, ..., statement)

    void continue() {
      try {
        translate_fork_body(block)
      } finally {
        this(new ForkDone());
      }
    }

    public react (ajava.runtime.ForkDone d) {
      FORK_WAITING_COUNT--;
      If (FORK_WAITING_COUNT <= 0) done();
    }
    void done() { OWNER_RECEPTIONIST <-- new ajava.runtime.ForkDone(); }

    declare_fork_actors(block, fork_idForkActor)
  }
```

These delegate actors contain the variables `MSG_WAITING_COUNT` and `FORK_WAITING_COUNT` which are initialized to the number of expression actor invocations, and the number of nested forks respectively. These decrement whenever a response is received from an expression actor, or a `ForkDone` message from a nested fork actor. When all of the responses have been received the ‘continue’ method is invoked containing the fork’s translated body and when this continuation and all child forks have completed it returns a `ForkDone` message to its parent fork actor. Each of these fork actors include a reactor for each of the possible response message type and assigns the result values to the correct fields. These reactors differentiate between invocations based on the unique request id numbers they were sent with.

```
translate_stmt(fork_statement(fork_id, statement_1,..., statement_n, block))
=
    fork_id _fork_id = new fork_id(OWNER_RECEPTIONIST);
    translate_fork_stmt(_fork_id, statement_1, 1)
    translate_fork_stmt(_fork_id, statement_2, 2)
    ...
    translate_fork_stmt(_fork_id, statement_n, n)

translate_expactor_call(fork_id, method_invocation_expression(id,
    arg_1, ..., arg_n), num)
=
    id <-- id.Request.create(fork_id, num, arg_1);
```

Fork blocks themselves translate directly as above, into the instantiation of the relevant delegate actor and a request message transmission for every actor expression invocation. These request messages define the caller as the delegate actor so that it receives the response messages.

Condition reactions

Condition reactions perform a reaction *when* a certain condition is true. They are not triggered when the condition becomes true, but whenever a reactor completes and the condition still holds. Condition reactors translate into standard reactors which end immediately if the condition is false.

```
translate_member(when_reactor_member(condition, reactor_number,
    block), actor_body)

protected class reactor_numberCondition {}

translate_member(reactor_member(
    reactor_numberCondition, msg,
    reactor_number, block))
```

At the end of every reaction a number of statements are inserted to evaluate the condition for each condition reactor and when true send a message to itself.

```
create_condition_checks(actor_body(member_1, ..., member_n))
=
    create_condition_checks(member_1)
    ...
    create_condition_checks(member_n);
```



```

create_condition_checks(when_reactor_member(condition,
                                             reactor_number, block))
=
  if ( condition ) this <-- new reactor_numberCondition();

```

Actor events

Actor events are public members which can have multiple subscriber actors to which it relays messages published. As the minimal language semantics allow public final actor fields, because they cannot change and can only be accessed by message passing, events are implemented as public final actors. Thus every event that is declared can be translated as follows:

```

translate_member(event_member(actor_interface, event_id), actor_body)
=
  public final Event<actor_interface> event_id
    = new Event<actor_interface>();

```

These event actors receive any message and multicast it to all their subscribers. An example implementation of an event actor can be found in Appendix F. Subscribing and unsubscribing actors is achieved using the “+=” and “-=” operators respectively, which translate into sending subscribe and unsubscribe messages.

```

translate_stmt(assignment_expression(op, lhs, rhs))
=
  If lhs is instanceof Event then:
    If op is “+=” then:
      lhs <-- new Event.Subscribe(rhs);
    Else if op is “-=” then:
      lhs <-- new Event.Unsubscribe(rhs);

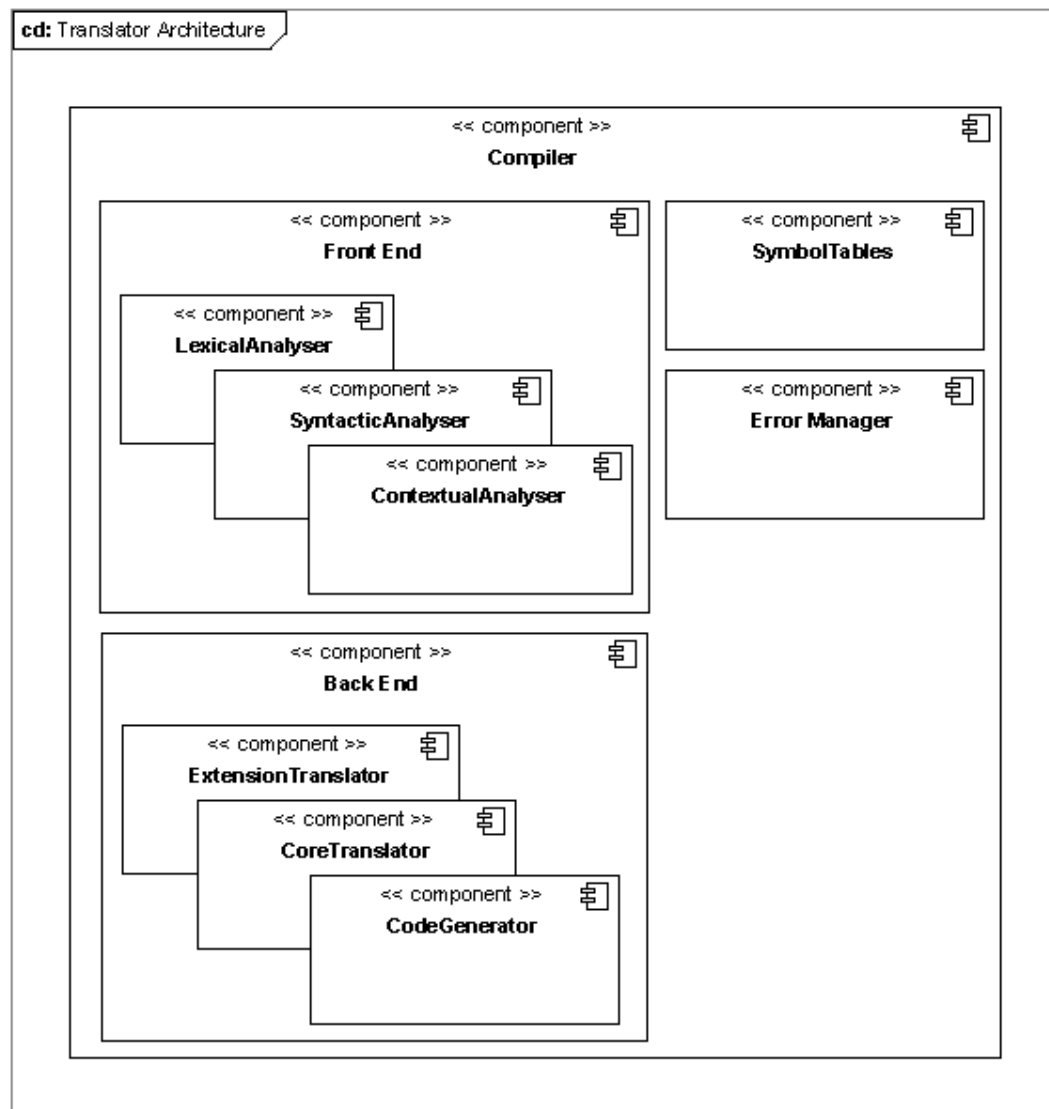
```

5 Translator Implementation

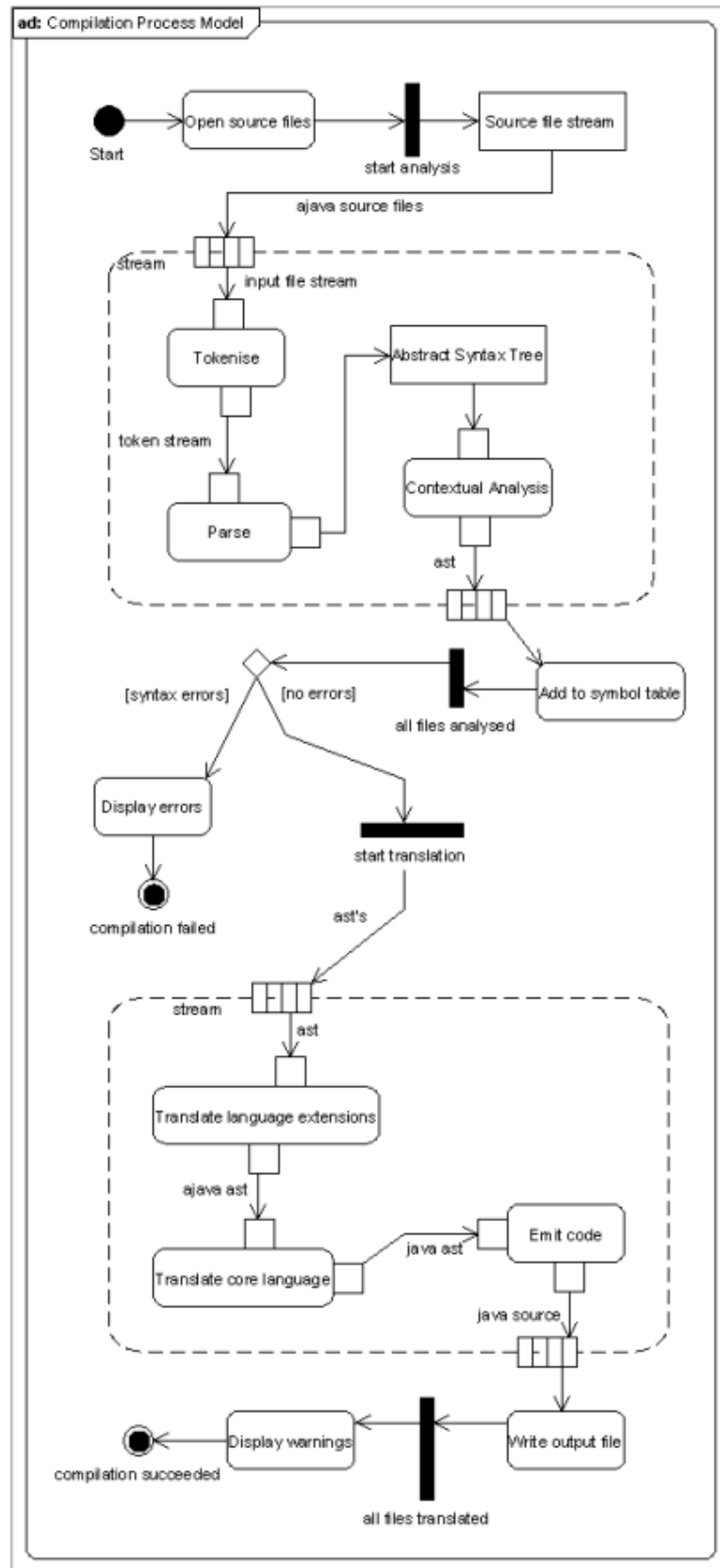
A prototype translator has been implemented which translate the actor language source files into Java source. It was developed using an iterative and incremental software development process as the system was naturally structured into clearly defined compilation phases that could be developed incrementally. This chapter describes the design, implementation and testing of this translator.

5.1 Translator Design

The system requirement is to take files written in the actor language and translate them into Java source which can be compiled by the Java compiler and executes with the correct behaviour. The system is for experimentation and demonstration purposes only and so is required to translate correct input into correct output only and is not required to consistently detect incorrect input.



[Figure 5: Translator architecture component diagram.]



[Figure 6: Translator process model diagram.]

The front end contains three sub-components: the lexical analyser which tokenises the input file into a stream of terminal symbols, the syntactic analyser which parses this stream into an abstract syntax tree, and the contextual analyser which decorates this tree with semantic information. The abstract syntax tree produced by the parser uses the Visitor pattern to allow the analysis and translation phases to traverse every node of the tree. Figure 6 demonstrates how this front end analysis stage must be repeated for each input file, such that symbol tables exist for each declared class before translation commences.

The back end takes a decorated syntax tree and progressively translates it into its Java. It then traverses the finished syntax tree serializing it into Java source code. As all of the language extensions can be translated into the minimal language, translation has been split into two stages as shown in figure 6: the first translating all of the language extensions into features of the minimal language, and the second translating the minimal language into plain Java. Translation of the minimal language can be achieved in a single pass, but translation of the language extensions requires the following passes:

1. Translates expression actors into normal actors with request and response messages.
2. Refactors all sequential actor invocations into nested fork blocks.
3. Translates fork blocks into delegate actor definitions.

5.2 Runtime Library

In addition to the translator itself a class library provides the foundation for the translated actor classes, and a means to bootstrap actor programs. The classes were designed as a by-product of the minimal language translation rules because their interfaces and behaviours were decided as the translation rules were codified.

The essential class is the `ActorBase` class, an abstract base class containing a message queue and the internal functions needed for an actor. An implementation of this class could have its own thread and a message loop which polls the queue for pending message, one advantage being that programs execute in a very predictable way. However this does command a large overhead and means that actors can never be garbage collected. Another implementation could use the `ExecutorService` class to provide a global thread pool which is referenced by every actor, such that whenever an actor has messages pending it requests some thread time and gets serviced by any free thread in the pool. This has the advantage that the program will only use an optimal number of operating system threads, but most importantly whenever an actor object is no longer referenced by any it can be garbage collected just like any other object. This implementation was used and is listed in Appendix E.

5.3 Translator Implementation

The translator was implemented in Java, using the JFlex and CUP tools as per the design. The parser takes a list of the source files (with the extension “.ajava”) on the command line and outputs a Java file with the same name for each. Implementation started with the lexical analysis, parsing and semantic analysis and then moving onto the

minimal language translator and code emission. The runtime class library was implemented as the translation phases were being developed, such that essential functionality was fully implemented before further extensions were attempted. Thus each phase was successively implemented, integrated and tested and the translations for the language extensions were left to the end. The complete prototype is implemented in 16,000 lines of Java code and additional parser generator and lexer source files with 116 non-terminals and 114 terminal symbols respectively.

As per the contingency plan some of the language extensions have been omitted from the implementation, such that a working cut-down translator exists. Support for condition reactors, actor events and the message shorthand have thus been omitted though the translator correctly handles `fork` blocks, expression actors and their invocations.

Furthermore the translator performs only very limited semantic checks due to the complexity of their implementation. To perform full type error checking it would have been necessary to open and parse Java `.class` and `.jar` files to access external class definitions and so the translator simply returns a warning if a type is referenced that cannot be resolved. This is not critical as the java compiler detects any type conflicts and syntax errors. Similarly semantic verification of linear types and concurrent forks have not been implemented. In short the prototype translator produced is quite sufficient to provide a platform for further investigation, but would not be suitable for commercial use.

5.4 Testing

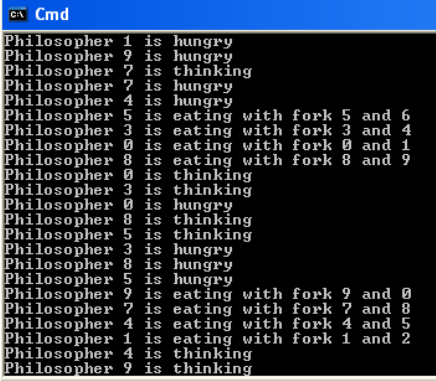
The translator is required to take a well formed actor language source file and output a well formed Java source file by correctly applying the language translation rules, but the translator's reaction to malformed input is not defined as a requirement and need not be tested. During development, the system was incrementally tested by taking well formed source files, and inspecting the translator's output for each. Once this develop/test cycle was complete, and the translator fully implemented, structured testing was performed, using programs that make use of the full language feature set. For one such program see Appendix E. These were translated and the resultant Java, compiled, executed, and any errors found were investigated and the relevant code corrected. The Java compiler itself served to detect many programming mistakes, as most bugs led to invalid syntax or semantics in the output files, rather than incorrect behaviour such that once the output compiled correctly it usually behaved correctly.

6 Evaluation and Findings

In order to evaluate the language, some example programs have been written, and tested using the prototype translator. The ease with which the programs could be written and the desired functionality achieved, provides a useful aid to the evaluation of the successfulness of the language, and the usefulness of its features.

6.1 Dining Philosophers

To evaluate the language's ability to provide clear and safe mechanisms to achieve complex inter-process communication, a solution to the classic dining philosopher's problem has been implemented (see Appendix F). A common solution to the problem is to only allow a philosopher to pick up both forks simultaneously so he can only pick them up if both neighbours aren't eating. In programming languages that use a shared-variable approach to IPC, a mutual exclusion semaphore must be carefully used to ensure that no two philosophers can simultaneously obtain or return forks, and a semaphore for each philosopher is used to block, until its neighbours make them available [36]. This method is far from transparent and so is prone to subtle but serious programming errors.



```
Cmd
Philosopher 1 is hungry
Philosopher 9 is hungry
Philosopher 7 is thinking
Philosopher 7 is hungry
Philosopher 4 is hungry
Philosopher 5 is eating with fork 5 and 6
Philosopher 3 is eating with fork 3 and 4
Philosopher 0 is eating with fork 0 and 1
Philosopher 8 is eating with fork 8 and 9
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 0 is hungry
Philosopher 8 is thinking
Philosopher 5 is thinking
Philosopher 3 is hungry
Philosopher 8 is hungry
Philosopher 5 is hungry
Philosopher 9 is eating with fork 9 and 0
Philosopher 7 is eating with fork 7 and 8
Philosopher 4 is eating with fork 4 and 5
Philosopher 1 is eating with fork 1 and 2
Philosopher 4 is thinking
Philosopher 9 is thinking
```

The message passing approach used in this language, vastly improves the understandability of the code, and is therefore less prone to errors. In the implementation a central 'table' actor keeps track of whether each philosopher is thinking, hungry or eating, and provides a common place to hold unused forks. This central actor acts like a monitor as no two messages can be received simultaneously and so has the same effect as the mutex semaphore. When a philosopher actor stops thinking, it sends an 'IsHungry' message to the table, which marks that philosopher as hungry, and if both forks are available passes them both to the philosopher, in a single `MoveForks` message. When the philosopher receives the forks it begins to eat, and when it finishes it passes them both back to the table. Thus instead of a critical region to ensure both forks are picked up at the same time, and a complex method using semaphores to block and wakeup waiting philosophers, philosophers receive both forks simultaneously in a single message, and return them in the same way.

Forks are implemented as linear objects, and so can be safely moved without fear of synchronization problems. On shared memory architectures these linear fork objects are passed by reference, giving the same performance that a semaphore based implementation would give, but with far better understandability. Furthermore the program could still be realised in a distributed way, as the fork passing is not reliant on shared variable access. Thus this language makes inter-process communication far more transparent, allows programs to be distributed without modification, and when shared memory is available gives the same performance as a semaphore based system.

6.2 *Parallel Quicksort*

The most common reason for using parallelism is when a large problem needs to be solved, parts of which can be performed in parallel. A simple example is the sorting of an array of integer values. The “Quicksort” algorithm does this using a divide-and-conquer technique to recursively partition a list, sorting the parts and merging them back together. Because of its divide and conquer nature, the algorithm naturally lends itself to parallel execution, as once partitioned any two parts may be sorted independently, and thus concurrently.

If an array was to be sorted in parallel on a distributed network, the array would have to be copied around, but as this programming language is aimed at shared memory architectures, an in-place algorithm is possible. A number of classes that police access to arrays have been implemented that allowing them to be temporarily divided up and merged back together efficiently, as the underlying array remains the same. In the actor language these classes are linear types. Using this virtual array distribution, two implementations of parallel in-place Quicksort have been written, one in plain Java and the other in the actor language, listed in Appendix F. Both implementations partition the array until the array is less than a given size, and then sorts the parts sequentially. The Java version does this by explicitly implementing a thread which spawns more threads, blocking until its child threads are complete. The actor version implements an expression actor which receives an array, partitions it, and invokes two new actors in parallel using a fork block.

Both implementations use the same Java helper functions but the actor implementation requires half the number of lines of code as the plain Java. This is due to the fact that Java provides no means to execute two functions in parallel. Doing this in Java requires coding a thread class with fields, and a constructor to hold any arguments, and further fields and accessors for any result. The expression actor, and fork constructs provide a much more concise means of doing this as they can be invoked with a parameter message, and return a result value. Furthermore unlike threads which once finished cannot be re-used, an actor can return a value and be invoked again and again indefinitely. These code overheads combined with the need to catch certain exceptions when using Java threads show the actor language can be much more concise.

Of course fewer lines of code, doesn’t imply faster execution. The implementations were benchmarked with arrays of varying sizes on a single core and a multi-core architecture, and the results are listed in Appendix F. The time taken to sort the same random array was compared using the different implementations. A small array of 10,000 elements was fastest using sequential quick sort on both architectures, but the actor implementation was still considerably faster than the Java implementation (see figure 7). It wasn’t until large arrays of a million elements that the parallel algorithms on the multi-core architecture began to really outperform the sequential. With an array of 1-million elements on a 6-core machine the sequential implementation took 1s to sort the array, the actor implementation took just over 0.6s, and the Java one just below 0.7s. This is clearly not a six-fold speedup although this is probably largely due to the large amount of memory access required in sorting, and because it was difficult to persuade the operating system’s scheduler to make the process run using multiple cores.

```
E:\Programming\Java\Eclipse Workspace\Comp3020 Compiler v1\tests>java -cp
_lang.jar;./program1/ org.taj.ajava.runtime.Runtime SortingBenchmark
Sorting Benchmark
Actors: 5ms
Threads: 26ms
Sequential: 3ms
```

[Figure 7: Parallel Quicksort for 10,000 elements run on single-core architecture]

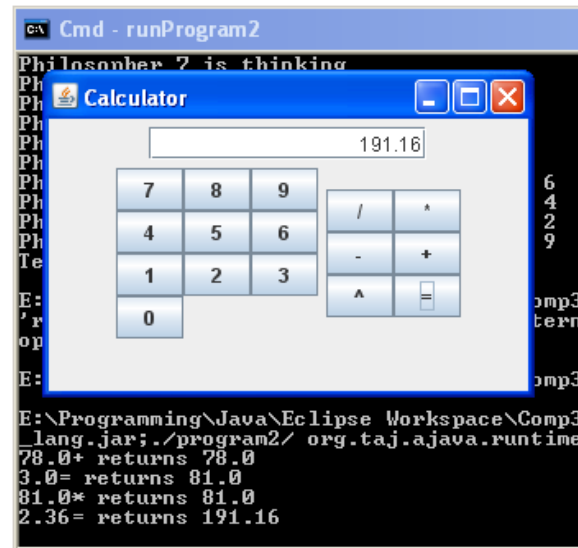
It was surprising that the actor implementation consistently outperformed Java threads. This is due to the fact that the actor runtime library used for the benchmarks used a thread pool and so it created far fewer threads, and those that were can be re-used. Furthermore there is no need to have countless threads sleeping or busy waiting for incoming messages, instead actors request call-backs only when they need a finite chunk of execution time. Thus the scheduling is handled implicitly; a far more appropriate model for a system with lots of regularly interacting agents. In this scenario the overhead to initialize, schedule and clean up the threads made a significant difference.

Finally this program demonstrates the power of linear types. The Java implementation provides no means to prevent programmers accidentally modifying the same array partition concurrently. Making these array objects linear means that this interference becomes impossible and is prevented at compile type rather than be discovered by intermittent erroneous behaviour.

6.3 Calculator App

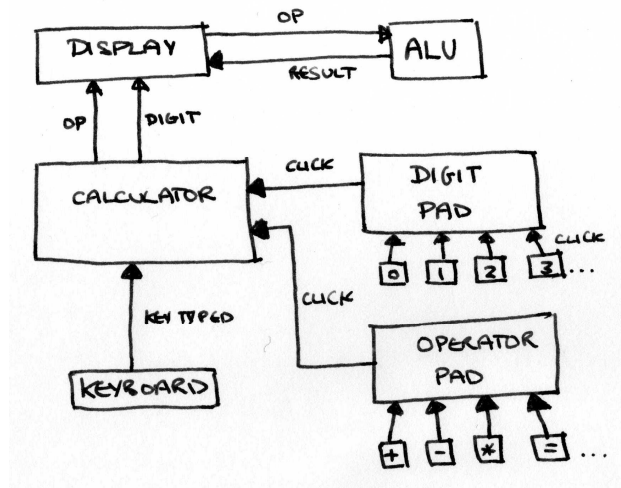
Programs are rarely batch computational tasks anymore, but are interactive applications, which may consist of many interacting modules and components. Thus, it is not sufficient to just demonstrate the suitability of a language to implement sorting algorithms and IPC but the language must be good at expressing interactive event driven applications.

To evaluate how well the language does this a small calculator program has been written which can be found in Appendix F. The application is split into a number of modules, each of which is implemented as an actor class (see figure 8). The whole application makes heavy use of event actors, even though the syntactic sugar for defining them implicitly was not implemented.



This program highlights some real strengths of the language and some areas to revise. One advantage was the ability to make modules very loosely coupled. In the example the calculator actor creates the other actors and when it does so subscribes the actors to events accordingly. This also allows programs to be easily extended as multiple actors can subscribe to an event. Even more powerful, is the ability to subscribe events to other

events, chaining them together. In this way modules can be very independent but can be linked up using events and common message types. This could even be programmed graphically, making event subscriptions like wires between actors like in Berkley's "Actor-oriented design" in Ptolemy II [24].



[Figure 8: Activity diagram showing modular structure of Calculator App]

Although the ability to flexibly link actors together proved to be a very natural way to implement modular applications, the expression actor extension had practical limitations. Expression actors were originally devised by Agha [8] to allow functions on arguments to be evaluated using a standard request-response protocol. This was shown to be useful in the sorting example, however for actors that represent persistent objects rather than function on arguments it is often necessary to query the state of multiple state variables separately. A future modification to the extended language should therefore allow different reactors to return different types, rather having one expression type that all reactors must return. This could be very simply implemented using multiple *Response* message types and would greatly improve usability. Some reactors could even be marked as `void` so that they could be invoked either synchronously or asynchronously.

6.4 Conclusion

The aim of this project was to begin the development of a language for implicitly parallel programming, using a natural and understandable notation that would be familiar to object oriented programmers, specifically for use on multi-core systems. This objective has been achieved, or even exceeded as the metaphor has been refined, a new language developed and a working prototype compiler implemented such that example programs have been compiled and executed using multiple processors.

Initial creative work demonstrated the usefulness of a language which defined agents which communicated by message passing, and research into the Actor model highlighted previous limitations. The model failed to allow the adequate sharing and locking of objects, but this has been overcome in the language using linear types. Thus unlike other Actor programming languages like SALSA [26], passive objects can be transferred between actors, without needing to be copied, a feature that has huge performance advantages that other message passing systems lack, as well as providing a neat way to avoid the use of synchronization constructs and mutual exclusion locks. As all inter-

actor communication is via message passing the interaction is far clearer, common synchronization bugs are eliminated, and programs could be distributed virtually without modification. In fact the use of linear types guarantees safety from interference problems, and forces correct coding in this respect at compile time.

The language also succeeds in providing a familiar environment for object oriented programmers. By building on Java and introducing “actor classes” the learning curve is dramatically reduced and defining actor behaviours declaratively using “reactor members” vastly improves the language’s clarity over other languages like ActiveCSharp [37] and Scala [38] that use message loops and blocking receive statements. It also defines clean interfaces for valid inter-process interaction and allows meaningful inheritance. This encourages modular design and dramatically reduces coupling. Actor events make this approach even more convenient as actors can define public outputs to which messages are published. Thus modules can be developed independently and interconnected externally much like linking wires between electronic components. As was found in the calculator example this leads to a loosely coupled system with high cohesion and a very clear structure as data flows between components.

Most importantly the language successfully provides means for implicit parallelism. The hybrid approach combining object oriented programming, with the actor model, and linear typing has resulted in a powerful and yet understandable language. Though languages that support various methods of implicit concurrency exist, they don’t tend to use such a familiar grammar, and are often aimed at large distributed computation rather than general purpose multi-threaded applications. Overall the language developed here provides a good solution to the increasing problem of programming multi-core architectures.

Two years ago a multidisciplinary group of Berkeley researchers met to discuss the implications of the emerging increase in parallel architectures and made 7 recommendations regarding future research in this field [28]. These included suggesting that the next generation of programming frameworks must be “human-centric”, “independent of the number of processors” and “naturally parallel” as architectures advance towards 1000s of cores per chip. The language developed in this project follows these principles and successfully contributes towards the achievement of a naturally parallel programming methodology. Most notably it has demonstrated that the union of the Actor model and linear typing provides a compelling new model for implicit concurrency. The hybrid is vastly more understandable and robust than shared memory communication but overcomes many of the inefficiencies of previous message passing systems. Though future revisions could enhance and refine the language, this project has successfully overcome previous limitations and has produced a new and innovative prototype to act as a springboard for further research.

6.5 Further Work

There is a large potential for further work arising from this project. As was previously discussed the extended language should be revised such that individual reactors can specify their own return types. On an implementation level the translator prototype could be refined and extended to perform semantic checking and to implement the remaining language.

A language concept that is worth consideration, but has been omitted for simplicity is the notion of an actor's locality. Real world objects don't just have connections to other objects, but they exist at locations proximate to other objects. The ability to send messages to all of an actor's neighbours for example, without needing individual references to each of them would be very useful for simulation especially.

Investigation could also be made into code optimization performing control flow analysis to serialize subsections of actor systems, translating them into sequential code. Programs could be written as actor systems, partitioned into independent actor groups, and each translated into optimized sequential code. Further research could be made into just-in-time compilation, such that programs could be "autotuned" [28] to the number of processors on the target system.

Finally further work into distributing computation would be helpful. Many grid frameworks exist but, making distribution an implicit part of a general purpose language would have huge advantages. This one language could be used to develop local applications, and distributed ones with a common syntax. Areas to investigate would include load balancing, and how to globally address actors irrespective of their locations. One interesting possibility would be to have 3 levels of actor identification: a global identifier, a locality within the program, and a network locator. Further work on this language could therefore produce a very useable and powerful notation for local multiprocessor systems and huge distributed systems alike, whilst providing competitive performance in both situations.

7 References

- [1] C. Hewitt, P. Bishop, and R. Steiger, *A Universal Modular ACTOR Formalism for Artificial Intelligence*, presented at International Joint Conference on Artificial Intelligence, Stanford, California, USA, 1973.
- [2] I. Greif, "Semantics of Communicating Parallel Processes," Ph.D. dissertation, MIT, Boston, Massachusetts, USA, 1975.
- [3] C. Hewitt, H. Baker, *Laws for Communicating Parallel Processes*, IFIP 77, pp. 987-992, North Holland, Amsterdam, 1977.
- [4] C. Hewitt, *Viewing control structures as patterns of passing messages*, Artificial Intelligence vol. 8, no. 3, pp. 323-364, June 1977.
- [5] W. D. Clinger, "Foundations of Actor Semantics," Ph.D. dissertation, MIT, Boston, Massachusetts, USA, 1981.
- [6] G. D. Plotkin, *A powerdomain construction*, SIAM J Computing 5, 3, September 1976, pp. 452-487.
- [7] G. Agha, C. Hewitt, "Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism," MIT, Boston, Massachusetts, USA, Tech. Rep. AIM-865, 1985.
- [8] G. Agha, *Actors: a model of concurrent computation in distributed systems*, Cambridge Boston MA USA: MIT Press, 1986, pp. 1-144.
- [9] G. Agha, *An Overview of Actor Languages*, ACM SIGPLAN Notices, vol. 21, no. 10, pp. 58-67, 1986.
- [10] G. Agha, "Relation between problems in large-scale concurrent systems and distributed databases," proceedings International Symposium on Databases in Parallel and Distributed Systems, 5-7 December 1988. pp. 2-12.
- [11] G. Agha, *Foundational issues in concurrent computing*, ACM SIGPLAN Notices, vol. 24, no. 4, pp. 60-69, 1989.
- [12] G. Agha, "The structure and semantics of actor languages," proceedings at Foundations of Object-Oriented Languages. REX School/Workshop, 28 May-1 June 1990, Noordwijkerhout, Netherlands. Berlin: Springer-Verlag, 1991, pp. 1-59.
- [13] G. Agha, "Distributed execution of actor programs," presented at Languages and Compilers for Parallel Computing. Fourth International Workshop, 7-9 Aug. 1991, Santa Clara, CA, USA. Berlin: Springer-Verlag, 1992, pp. 1-17.
- [14] G. Agha, I. A. Mason, S. F. Smith, C. L. Talcott, "Towards a Theory of Actor Computation," proceedings of the Third International Conference on Concurrency Theory 1992. London: Springer-Verlag, 1992, pp. 565-579.
- [15] W. Kim, G. Agha, "Compilation of a highly parallel actor-based language," Languages and Compilers for Parallel Computing. 5th International Workshop Proceedings, 3-5 Aug 1992, New Haven, CT, USA. Berlin: Springer-Verlag, pp. 1-15.
- [16] G. Agha, C. J. Callsen, "ActorSpace: An Open Distributed Programming Paradigm," Proceedings 4th Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices, 1993, pp. 23-323.
- [17] G. Agha, S. Frolund, W. Y. Kim, R. Panwar, A. Patterson, D. Sturman, *Abstraction and modularity mechanisms for concurrent computing*, IEEE Parallel & Distributed Technology: Systems & Applications, vol. 1, no. 2, pp. 3-14, May 1993.

- [18] G. Agha, *Formal methods for actor systems: a progress report*, IFIP Transactions C (Communication Systems), vol. C-10, 1993, pp. 217-228.
- [19] G. Agha and I. A. Mason, S. F. Smith, C. L. Talcott, *A Foundation for Actor Computation*, *Journal of Functional Programming*, vol. 7, no. 1, pp. 1-72, 1997.
- [20] C. A. Varela, G. A. Agha, "What after Java? From objects to actors," *Proceedings of the Seventh International Conference on World Wide Web*, 1998, pp. 573-577.
- [21] C. Varela and G. Agha. "Programming Dynamically Reconfigurable Open Systems with SALSA," *OOPSLA 2001 Intriguing Technology Track*. ACM SIGPLAN Notices vol. 36, no. 12, December 2001, pp. 20-34.
- [22] J. W. Janneck, "Actors and their Composition", *Formal Aspects of Computing*, vol. 15, no. 4, pp. 349-369, Dec 2003.
- [23] E. A. Lee, S. Neuendorffer, M. J. Wirthlin, "Actor-Oriented Design of Embedded Hardware and Software Systems", *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 231-260, 2003.
- [24] E. A. Lee, S. Neuendorffer, "Classes and subclasses in actor-oriented design," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, Jun 2004, pp. 161-168.
- [25] E. A. Lee, "The Problem with Threads," *Computer*, vol. 39, no. 5, pp. 33-42, May 2006.
- [26] C. A. Varela, G. Agha, .W. J. Wang, T. Desell, K. E. Maghraoui, J. LaPorte, A. Stephens, "The SALSA Programming Language 1.1.2 Release Tutorial", rpi.edu, pp. 13, Rensselaer Polytechnic Institute, Troy, New York, Feb 2007 [Online]. Available: <http://wcl.cs.rpi.edu/salsa/tutorial/salsa112v.pdf> [Accessed 8 Jan, 2008].
- [27] Valve Corporation, "Hardware Survey Results", May 3rd 2008 [Online]. Available: <http://www.steampowered.com/status/survey.html> [Accessed 3 May, 2008].
- [28] K. Asanovic et al. "The Landscape of Parallel Computing Research: A View from Berkeley", University of California, Berkeley, Tech Rep. UCB/EECS-2006-183. December 18, 2006. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf> [Accessed 3rd May 2008].
- [29] L. J. Flynn. "Intel Halts Development of 2 New Microprocessors", *The New York Times*, May 8th 2004. Available: <http://www.nytimes.com/2004/05/08/business/08chip.html?ex=1399348800&en=98cc44ca97b1a562&ei=5007> [Accessed May 3rd, 2008].
- [30] J. Y. Girard, "Linear Logic," *Theoretical Computer Science*, vol. 50, pp. 1-102, 1987.
- [31] P. Wadler, "Linear types can change the world!", in *Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, pp. 347-359, 1990.
- [32] J. Basney and R. Raman and M. Livny, "High Throughput Monte Carlo," in *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.
- [33] J. R. Gurd et al, "Fine-grain parallel computing: the dataflow approach," in *Proceedings of an advanced course on Future parallel computers*, Pisa, Italy, pp. 128, 1987.
- [34] K. M. Chandy, "Event-Driven Applications: Costs, Benefits and Design Approaches," California Institute of Technology, 2006.

- [35] K. M Chandy, C. Kesselman, “Compositional C++: Compositional Parallel Programming,” California Institute of Technology, Pasadena, CA, USA, Tech. Rep. 105, 1992.
- [36] A. S. Tanenbaum, *Modern Operating Systems 2nd Ed.*, India: Prentice-Hall, 2001.
- [37] R. Güntensperger, J. Gutknecht, “Active C#,” in *Proceeding of International .NET Technologies Workshop*, Plzen, Czech Republic, 2004. Available: <http://www.avocado.ethz.ch/activecsharp/ActiveCSharp.pdf> [Accessed: May 3rd, 2008].
- [38] M. Odersky, “The Scala Language Specification”, v. 2.7, Programming Methods Laboratory, EPFL, Switzerland, May 5th 2008. Available: <http://www.scala-lang.org/docu/files/ScalaReference.pdf> [Accessed: May 3rd, 2008].

Appendix A.

Initial Syntax Experiment

The example below introduces the basic concepts of the language but was designed prior to research on the Actor model, and so contains “observers” (the view keyword) which are not appropriate in potentially distributed parallel computing. This example was an experiment looking at whether user interactive applications are better written using this model, due to their high emphasis on events.

```
// Calculator System
// -----
// a demo causal system, describing a pocket
// calculator program. it must be noted that this describes
// a system, with interactions between objects, rather than
// an explicit algorithm. there is a freedom into how and where,
// the object's are stored and executed which allows distribution,
// and concurrency.

// a notification definition, used throughout the
// system. details a single digit in the range 0..9
notification DigitNotification: NotificationBase
{
    int Digit;
}

// NumberBox object class definition. contains a complete
// definition of the object's type, with instance and state variables,
// externally viewable observers, constructors, and reactions to notifications.
class NumberBox: TextBox
{
    // notification definitions. these are public
    // but are defined here as they are used by this class
    // and it's ancestors
    notification SetNumber { double number; }
    notification Clear {}
    notification AppendDot {}

    // this is a state variable which,
    // can be changed in reaction to notifications,
    // and is private like all instance variables.
    state double number;

    // an observer, for externally viewing a state
    // or instance member.
    view double Number { return number; }

    // a constructor
    cons (double number) {
        base(Double.ToString(number));
    }

    // reactions
    // maps causes to effects

    // reacts to set number notifications
    react (SetNumber) {
        // send a new TextBox.SetText notification to "this"
        TextBox.SetText(text=Double.ToString(e.number)) -> this;
    }

    // reacts to a digit notification, by appending the digit
    // to the current number.
    react (DigitNotification) {
        SetNumber(number = (this.number * 10) + e.digit) -> this;
    }

    // clears the number to 0.0
    react (Clear) {
```

```

        SetNumber(number=0.0) -> this;
    }

    // reaction execution:
    // after all reactions from ancestors are executed
    // each of their individual changes are applied
    // and the object becomes observably different, and events
    // are raised, so the reaction seems atomic.
}

// A button with a single digit on it.
class DigitButton: Button
{
    // cannot be changed after constructor
    final int digit;

    // provides a view, to observe the digit
    // member. observer's cannot modify the objects
    // state or raise any events; though they may invoke
    // other observers and static methods.
    view int Digit { return digit; }

    // creates the button
    cons (int digit) {
        // init base (using static method)
        base(Int.ToString(digit));

        // set the digit
        this.digit = digit;

        // subscribe to Clicked events
        this.Clicked += this;
    }

    // event (DigitNotification), raised when the button is clicked
    // and details the digit clicked (same as Digit).
    event DigitClicked;

    // reacts to clicked notifications (which say that this has
    // been clicked) from itself, by raising the DigitClicked event.
    react (ClickedNotification) {
        // if this event is from this object
        if (e.Sender == this) {
            // creates and inits a new digit notification, and
            // raises the DigitClicked event with it.
            DigitNotification(Digit=this.digit) -> DigitClicked;
        }
    }
}

// A set of ten digit buttons from 0..9.
class DigitButtons: Panel
{
    // instance variable. stays the same for an
    // instance once it has been set in the constructor
    final DigitButton[] buttons;

    // parameterised observer
    view DigitButton Button[int digit] { return buttons[digit]; }

    // constructor
    cons () {
        // make buttons
        buttons = new DigitButton[]();

        // forevery does not enforce order
        // does every in a list. 0:1:9 - shorthand for
        // 0,1,2,3,4,5,6,7,8,9 - start:step:end like in matlab.
        forevery (int digit in 0:1:9) {
            // creates the button
            DigitButton btn = new DigitButton(i);
            btn.DigitClicked += this;
            buttons[digit] = btn;

            // call an init method
            // to add the widget to the panel

```



```

        this.AddWidget(btn);
    }

    }

    // an event that raises a DigitNotification can be subscribed to which
    // is raised when any digit button is pressed. events are untyped, so that
    // any notifications can be passed to them.
    event DigitClicked;

    // a reaction to a digit notification
    // (just passes it to the DigitClicked event)
    // could have just subscribed the event for shorthand).
    react (DigitNotification) {
        e -> DigitClicked;
    }
}

// notification sent when an operator button is clicked
notification OperatorNotification
{
    string Operator;
}

// an operator button
class OpButton: Button
{
    // the operator character +,-,= etc...
    final string operator;

    // raised when the button is clicked.
    event OperatorClicked;

    // constructor
    cons (string op) {
        base(op);
        this.operator = op;
        this.Clicked += this;
    }

    // when clicked, raise operator clicked event.
    react (ClickNotification) {
        OperatorNotification(Operator=this.operator) -> OperatorClicked;
    }
}

// A panel with digit buttons, arithmetic operators
// on it, and an evaluation button on it.
class ButtonPad: Panel
{
    // instance variables, which can be
    // set in the constructor, but which are
    // immutable once the object has been created
    final DigitButtons digits;
    final OpButton plus, minus, equals;

    // raised when any of the buttons on the pad
    // is clicked.
    event ButtonClicked;

    // the default constructor
    cons () {
        // create children
        digits = new DigitButtons();
        plus = new OpButton("+");
        minus = new OpButton("-");
        equals = new OpButton("=");
        this.AddWidget(digits);
        this.AddWidget(plus);
        this.AddWidget(minus);
        this.AddWidget(equals);

        // subscribe an event to button pressed events
        // so that notifications, are published to all
        // who subscribe to that event
        digits.DigitClicked += ButtonClicked;
        plus.Clicked += ButtonClicked;
        minus.Clicked += ButtonClicked;
    }
}

```

```

        equals.Clicked += ButtonClicked;
    }
}

// arithmetic logic unit, that actually carries out the
// computation. (overkill but included for neatness and
// demonstration of good system design).
class ALU
{
    // a request for an arithmetic logic
    // function to be applied
    notification Request
    {
        double A,B;
        string Operator;
    }

    // the result is computed and this notification,
    // containing the result is returned to the request
    // notification sender.
    notification Result
    {
        double Value;
    }

    // react to an ALOperation, computing the
    // result and replying to the sender
    react (ALU.Request) {
        // init
        double result = e.A;

        // apply operation
        if (e.operation == "+") result += e.B;
        else if (e.operation == "-") result -= e.B;

        // reply to sender
        ALU.Result(Value=result) -> e.Sender;
    }
}

// A panel with a keypad, and number display.
class CalculatorPanel: Panel
{
    // permanent members
    final ButtonPad buttons;
    final NumberBox number;
    final ALU alu;

    // state variables
    state string operator;
    state double mem;

    // the default constructor
    cons () {
        // init state
        mem = 0.0;
        operator = "+";

        // create children
        number = new NumberBox();
        buttons = new ButtonPad();
        alu = new ALU();

        // call init procedures
        this.AddWidget(number);
        this.AddWidget(buttons);

        // listen to button presses
        buttons.ButtonClicked += this;
    }

    // reactions - mapping between causes and effects,
    // notifications and responses.

    // when a button is pressed, append to number display
    react (DigitNotification) {
        e -> number;
    }
}

```

```

    }

    // when an operator is clicked
    react (OperatorNotification) {
        if (e.Operator == "=") {
            // compute result
            ALU.Request(A=mem, B=number.Number, Operator=e.Operator) -> alu;
        } else {
            // remember number and operator
            mem = number.Number;
            operator = e.Operator;

            // clear number display
            NumberBox.Clear() -> number;
        }
    }

    // when the result has been computed, show in display
    react (ALU.Result) {
        NumberBox.SetNumber(number=e.Value) -> number;
    }
}

// the main class. an instance of this is created
// using the default constructor, when the system
// starts. this contains the causal system which is
// then simulated.
main class CalculatorApp
{
    // the actual calculator UI.
    final CalculatorPanel calcPanel

    // constructor
    cons () {
        calcPanel = new CalculatorPanel();
    }
}

```

Appendix B.

Resource sharing experiments

The examples below investigate ways in which mutual exclusion, and resource sharing can be achieved in actor systems, without introducing any further concepts so that all resources must be actors.

```
// 2 Queue Example
// -----

actor Queue
{
    // queue object (not active object)
    state QueueImplementation queue;

    cons () {
        queue = new QueueImplementation();
    }

    notification enqueue { Object o }
    notification dequeue { Actor a }
    notification dequeue_reply { Object o }
    notification dequeue_to { Queue q }

    // add to queue
    react (enqueue) {
        this.queue.pushtail(o=e.o);
    }

    // remove from queue
    react (dequeue) {
        Object v = this.queue.pophead();
        dequeue_reply(o=v) -> e.a;
    }

    // move from here to another queue
    react (dequeue_to) {
        Object v = this.queue.pophead();
        enqueue(o=v) -> e.q;
    }
}

actor Demo
{
    Queue q1, q2;

    cons () {
        q1 = new Queue();
        q2 = new Queue();
    }

    react (init) {
        Queue.enqueue( 1234 ) -> q1;
        Queue.dequeue_to( q2 ) -> q1;
    }
}

// Problem: In some cases it might matter that there is a
// period whilst queue 1, and queue 2 are both empty
// (while message is in transit)

// 2 Queue Example - with locking
// -----

actor Queue
{
    // queue object (not active object)
    final QueueImplementation queue;

    // indicates is not currently readable
```

```

state boolean locked;
state NotificationBuffer buffer;

cons () {
    queue = new QueueImplementation();
    buffer = new NotificationBuffer();
    locked = false;
}

notification enqueue { Object o; Actor rsvp; }
notification enqueue_done {}
notification dequeue { Actor a }
notification dequeue_reply { Object o }
notification dequeue_to { Queue q }

// add to queue
react (enqueue) {
    // still allowed to enqueue, even when
    // read locked.
    this.queue.pushtail(e.o);
    enqueue_done() -> e.rsvp;
}

// when has been added to other enqueue has done
react (enqueue_done) {
    if (locked) {
        // unlock and process buffered messages
        locked = false;
        for-all-in(buffer) -> this;
    }
}

// remove from queue
react (dequeue) {
    if (!locked) {
        Object v = this.queue.pophead();
        dequeue_reply(o=v) -> e.a;
    } else {
        // if locked, buffer request
        buffer.addmessage(e);
    }
}

// move from here to another queue
react (dequeue_to) {
    if (!locked) {
        Object v = this.queue.pophead();
        enqueue(o=v, rsvp=this) -> e.q;
        locked = true;
    } else {
        // if locked buffer request
        buffer.addmessage(e);
    }
}
}

actor Demo
{
    Queue q1, q2;

    cons () {
        q1 = new Queue();
        q2 = new Queue();
    }

    react (init) {
        Queue.enqueue( 1234 ) -> q1;
        Queue.dequeue_to( q2 ) -> q1;
    }
}

// Solution: Buffers queue 1's read (dequeue) requests, whilst queue
// 1 is inconsistent. Then when confirmation comes that it is added
// to queue 2, we process buffered messages.
// Advantages: Only queue 1 needs to be locked, to prevent inconsistency.
// Even whilst locked we can enqueue onto queue 1, we just cant

```

```

//          dequeue

Producer/Consumer Actor Example
-----
// cheats somewhat as the message buffering removes
// any need for an explicit buffer, though one can be
// implemented easily enough, and both producer and
// consumer communicate via it.

actor Producer
{
    cons () { }

    notification next { Consumer c } // requests next element
    notification element { Producer p; Object value; } // sends an element

    react (next) {
        // produce new element, and send to
        // the consumer
        element(p=this,value=new Element()) -> e.c;
    }
}

actor Consumer
{
    cons () {}

    // starts consuming from this producer
    notification add_producer { Producer p; }

    react (add_producer) {
        // send 10 "next" messages to producer
        // (maintains buffer of 10 elements therefore)
        Consumer.next(c=this) x 10 -> e.p;
    }

    react (Producer.element) {
        // consume it... yum yum yum
        // send request for another
        Consumer.next(c=this) -> e.p;
    }
}

main actor PCDemo
{
    final Producer p1, p2;
    final Consumer c1, c2, c3;

    cons() {
        p1 = new Producer();
        p2 = new Producer();
        c1 = new Consumer();
        c2 = new Consumer();
        c3 = new Consumer();
    }

    // occurs on startup, after cons
    react (init) {
        // add producer 1 to consumers
        Consumer.add_producer(p=p1) -> c1, c2, c3;
        // consumer 1, can also consume from p2
        Consumer.add_producer(p=p2) -> c2;
    }
}

// Alternative using Semaphores
// -----

semaphore mutex = 1
semaphore full = 0
semaphore empty = BUFFER_SIZE

procedure producer() {
    while (true) {
        item = produceItem()
        down(empty)
    }
}

```

```

        down(mutex)
        putItemIntoBuffer(item)
        up(mutex)
        up(full)
    }
}

procedure consumer() {
    while (true) {
        down(full)
        down(mutex)
        item = removeItemFromBuffer()
        up(mutex)
        up(empty)
        consumeItem(item)
    }
}

// Alternative using Monitors
// -----

monitor ProducerConsumer {

    int itemCount
    condition full
    condition empty

    procedure add(item) {
        while (itemCount == BUFFER_SIZE) {
            wait(full)
        }

        putItemIntoBuffer(item)
        itemCount = itemCount + 1

        if (itemCount == 1) {
            notify(empty)
        }
    }

    procedure remove() {
        while (itemCount == 0) {
            wait(empty)
        }

        item = removeItemFromBuffer()
        itemCount = itemCount - 1

        if (itemCount == BUFFER_SIZE - 1) {
            notify(full)
        }

        return item;
    }
}

procedure producer() {
    while (true) {
        item = produceItem()
        ProducerConsumer.add(item)
    }
}

procedure consumer() {
    while (true) {
        item = ProducerConsumer.remove()
        consumeItem(item)
    }
}

```

Resources as Actors and Linear Types

The examples below compare solutions to common resource sharing problems using the possible approaches: variable locking, resources as actors and resources as linear types.

```
public class SharedResource {
    Actor currentActor;
    Queue grabQueue = new Queue();

    class Ready { SharedResource resource; }
    class GrabResource { Actor sender; }
    react (GrabResource msg) {
        if (currentActor == null) {
            currentActor = msg.sender;
            msg.sender <-- new Ready(this);
        } else {
            grabQueue.enqueue(msg.sender);
        }
    }

    class ReleaseResource { Actor sender; }
    react (ReleaseResource msg) {
        if (currentActor == msg.sender) {
            currentActor = null;
            this.grabQueue.dequeue();
        } else {
            grabQueue.remove(msg.sender);
        }
    }

    class UseResource { Actor sender; }
    react (UseResource msg) {
        if (currentActor == sender) {
            ...
        }
    }
}
```

```
public linear class SharedResource {
}

public class ResourceUser {
    class SendResource { Actor sender; SharedResource value; }
    react (SharedResource res) {
        // use resource
        ...
        // return to sender
        res.sender <-- res.value;
    }
}
```


Appendix C.

This appendix lists the full grammar of the language, in Backus-Naur form. Bold names are terminal symbols, square brackets signify optional sections, and curly braces boundless repetition. The grammar for the minimal language is presented, followed by the extended language.

Minimal Language Grammar

```
type_declaration      ::= modifiers ( actor_class | actor_interface
                                | java_class | java_interface )

actor_class           ::= ( aclass | actor ) identifier
                        [ extends data_type ]
                        [ implements data_type_list ]
                        actor_class_body

actor_class_body      ::= "{ " { modifiers actor_class_member } "}"

actor_class_member    ::= react "(" data_type identifier ")"
                        statement_block |

                        type_declaration |

                        data_type
                        field_declarator { ",", field_declarator }
                        ";" |

                        identifier "(" expression_list ")"
                        constructor_block |

                        data_type identifier
                        "(" parameter_list ")" statement_block

actor_interface       ::= ainterface identifier
                        [ extends data_type_list ]
                        actor_interface_body

actor_interface_body  ::= "{ " { modifiers actor_interface_member }
                        "}"

actor_interface_member ::= react "(" data_type identifier ")" ";" |

                        type_declaration |

                        data_type
                        field_declarator { ",", field_declarator }
                        ";" |

statement_block       ::= "{ " { block_statement } "}"

block_statement       ::= local_variable_declaration |

                        statement

local_variable_declaration ::= [ final ] data_type
                                variable_declarator_list ";"

statement             ::= ";" |

                        statement_block |

                        if_statement |
```

```

switch_statement |
return_statement |
break_statement |
expression_statement

if_statement ::= if "(" expression ")" statement
               [ else statement ]

switch_statement ::= switch "(" expression ")"
                   "{" { switch_group } "}"

switch_group ::= switch_label { switch_label }
               { statement }

switch_label ::= case expression ":" |
                default ":"

return_statement ::= return expression ";"

break_statement ::= break ";"

expression_statement ::= expression ";"

expression ::= expression1 [ assignment_op expression1 ]

expression1 ::= expression2
              [ "?" expression ":" expression1 ]

expression2 ::= expression3 [ instanceof data_type |
                              { infix_op expression3 } ]

expression3 ::= prefix_op expression3 |
               primary_expression postfix_op |
               "(" data_type ")" expression3

primary_expression ::= literal |
                    new type_name class_instance_creator |
                    expression4

class_instance_creator ::= arguments |
                        { array_indexer }
                        [ [] { [] } array_initializer ]

array_indexer ::= "[" expression "]"

array_initializer ::= "{" { expression | array_initializer } "}"

expression4 ::= "(" expression ")" |
              secondary_expression |
              expression4 array_indexer |
              expression4 "." secondary_expression

secondary_expression ::= identifier [ arguments ] |
                     this [ arguments ] |

```

	super [arguments]
arguments	::= "(" [expression { ",", expression }] ")"
assignment_op	::= "=" <-- += -= etc...
infix_op	::= "+" "-" "*" "/" == && etc...
prefix_op	::= ++ -- "!" etc...
postfix_op	::= ++ --

Extended Language Grammar

type_declaration	::= modifiers (actor_class actor_interface message_type java_class java_interface)
message_type	::= message identifier "(" parameter_list ");"
actor_class	::= (aclass actor) identifier [extends data_type] [implements data_type_list] [returns data_type] actor_class_body
actor_class_body	::= "{" { modifiers actor_class_member } "}"
actor_class_member	::= react "(" data_type identifier ")" statement_block react-when "(" expression ")" statement_block type_declaration data_type field_declarator { ",", field_declarator } ";" identifier "(" expression_list ")" constructor_block data_type identifier "(" parameter_list)" statement_block event type_name identifier { ",", identifier } ";"
actor_interface	::= ainterface identifier [extends data_type_list] actor_interface_body
actor_interface_body	::= "{" { modifiers actor_interface_member } "}"
actor_interface_member	::= react "(" data_type identifier)" ";" type_declaration data_type field_declarator { ",", field_declarator } ";"

```

event type_name identifier
{ "," identifier } ";"

statement_block ::= "{" { block_statement } "}"

block_statement ::= local_variable_declaration |
                    statement

local_variable_declaration ::= [ final ] data_type
                               variable_declarator_list ";"

statement ::= ";" |
            statement_block |
            if_statement |
            switch_statement |
            return_statement |
            break_statement |
            expression_statement |
            fork_statement

if_statement ::= if "(" expression ")" statement
               [ else statement ]

switch_statement ::= switch "(" expression ")"
                   "{" { switch_group } "}"

switch_group ::= switch_label { switch_label }
               { statement }

switch_label ::= case expression ":" |
                default ":"

return_statement ::= return expression ";"

break_statement ::= break ";"

fork_statement ::= fork "(" { fork_exp } ")"
                  [ statement_block | ";" ] |
                  fork statement_block

fork_exp ::= local_variable_declaration |
             expression_statement

expression_statement ::= expression ";"

expression ::= expression1 [ assignment_op expression1 ]

expression1 ::= expression2
              [ "?" expression ":" expression1 ]

expression2 ::= expression3 [ instanceof data_type |
                              { infix_op expression3 } ]

expression3 ::= prefix_op expression3 |

```

```

primary_expression postfix_op |
    "(" data_type ")" expression3

primary_expression ::= literal |
                    new type_name class_instance_creator |
                    expression4

class_instance_creator ::= arguments |
                        { array_indexer }
                        [ [] { [] } array_initializer ]

array_indexer ::= "[" expression "]"

array_initializer ::= "{ { expression | array_initializer } }"

expression4 ::= "(" expression ")" |
              secondary_expression |
              expression4 array_indexer |
              expression4 "." secondary_expression

secondary_expression ::= identifier [ arguments ] |
                       this [ arguments ] |
                       super [ arguments ]

arguments ::= "(" [ expression { "," expression } ] ")"

assignment_op ::= "=" | <-- | += | -= | etc...

infix_op ::= "+" | "-" | "*" | "/" | == | && | etc...

prefix_op ::= ++ | -- | "!" | etc...

postfix_op ::= ++ | --

```

Appendix D.

This appendix lists the formal translation rules which describe how the minimal language translates into Java, and how the language extensions can be translated into the minimal language.

Minimal Language Translation Rules

```
f0(type_declaration(modifier_1, ..., modifier_n, class_definition))
=
    f12(modifier_1)
    ...
    f12(modifier_n)

    f1(class_definition)
```

Translates irrelevant modifiers (ignoring “linear”) and translates the class definition.

```
f1(actor_class(name, is_singleton, body, parent,
               interface_1, ..., interface_n))
=
    class name
    If parent == null: extends ajava.lang.Actor
    Else:             extends parent
    implements interface_1, ..., interface_n
    f2(body, singleton, name)
```

Translate actor class definition into passive java class definition. If no parent is specified inherits from ajava.lang.Actor.

```
f2(actor_class_body(member_1, ..., member_n), is_singleton, name)
=
    {
        f3(member_1)
        f3(member_2)
        ...
        f3(member_n)

        protected void processMessage(ActorMessage msg) {
            switch (msg.reactorId) {
                For every member that is a reactor:
                f4(member_1)
                f4(member_2)
                ...
                f4(member_n)
                default: super.processMessage(msg); return;
            }
        }

        If ‘is_singleton’ (i.e. keyword was “actor” not “aclass”) then:
        private static class SingletonHolder {
            private final static name instance = new name();
        }
        public static name getInstance() {
            return SingletonHolder.instance;
        }
    }
```

```

    }
}

```

Translates an actor class body, into a passive java class body. Translates each member directly, then creates a processMessage method to invoke the correct reactor for any buffered message, and if it is an actor singleton, includes a static instance.

```

f3(reactor_member(message_type, message_id, reactor_number, block))
=
    public void deliver(message_type message_id) {
        bufferMessage(new ActorMessage(message_id, reactor_number));
    }

    protected void react(message_type message_id)
    f5(block)

```

Translates a reactor into a private method, and a public deliver method

```

f4(reactor_member(message_type, message_id, reactor_number))
=
    case reactor_number:
        react((message_type)msg.payload);
    return;

```

Emmits one of the case statements for the processMessage method, such that buffered messages invoke the correct reactor.

```

f5(statement_block(statement_1, ..., statement_n))
=
    {
        f6(statement_1)
        f6(statement_2)
        ...
        f6(statement_n)
    }

```

Translates every statement, in a statement block into a java statement.

```

f6(statement)
=

```

Java statements and expressions, other than loops (for, while, do) and the try-catch-finally constructs are translated without modification. However assignment expressions that use the '<--' operator, are translated using f7.

```

f7(transfer_expression(lhs, rhs))
  If lhs is an actor.
=
  If rhs is a basic type OR rhs is a class creation expression:
    lhs.deliver(rhs);

  Else if rhs is of a linear type
    lhs.deliver(rhs);
  If rhs is an identifier expression
    rhs = null;

  Else
    lhs.deliver(rhs.clone());

  Else, if lhs is an instance of a linear type:
=
  lhs = rhs;
  If rhs is an identifier expression (make it a destructive read)
    rhs = null;

```

Translates message send, and linear object transfer expressions.

```

f7(identifier_expression(id, parent_expression))
=
  If parent_expression.id refers to a singleton actor class:
    parent_expression.id.getInstance()
  Else
    parent_expression.id

```

Translates an identifier expression, such that if it refers to the class name of a singleton actor, it invokes its accessor method, and otherwise it translates directly.

```

f1(actor_interface(name, body, parent_1, ..., parent_n))
=
  interface name extends parent_1, ..., parent_n
  f9(body)

```

Translates an actor interface into a java interface.

```

f9(actor_interface_body(member_1, ..., member_n))
=
  {
    f10(member_1)
    f10(member_2)
    ...
    f10(member_3)
  }

```

Translates an actor interface body, into a java interface body.


```
f10(reactor_signature(message_type, message_id))
=
    void deliver(message_type message_id);
```

Translates a reactor signature, into a deliver method signature.

```
f12(modifier)
=
```

*If modifier != "linear" then:
modifier*

Ignores the linear modifier for the purposes of translation. This modifier is just used for semantic analysis purposes.

Extension Translation Rules

```
translate(actor_class(name, is_singleton, body, parent, return_type,
    interface_1, ..., interface_n))
```

```
=
```

```
    class name extends parent
    implements interface_1, ... interface_n
    translate_actor_body(body, return_type, singleton, name)
```

Translates an extended language actor class (which may define a return type) into a minimal language actor class.

```
translate_actor_body(actor_class_body(member_1, ..., member_n),
    return_type, is_singleton, name)
```

```
=
```

```
{
```

If return_type != null (i.e. is an expression actor) then:

```
    public static class Response extends ActorResponseMessage {
        public return_type value;
        private Response(final ActorRequestMessage request)
        {
            super(request);
        }
    }
```

```
    public static class Request {
        declare_request_creator(member_1)
        ...
        declare_request_creator(member_n)
    }
```

```
    translate_exp_member(member_1)
    translate_exp_member(member_2)
    ...
    translate_exp_member(member_n)
```

Else (i.e. if not an expression actor) then:

```
    translate_member(member_1)
    translate_member(member_2)
```

```

    ...
    translate_member(member_n)
}

```

Translates an extended language actor body, into a minimal language actor body. If an actor return type is defined the declares Response and Request message classes, or otherwise just translates straight.

```

declare_request_creator(reactor_member(message_type))
=
public static message_typeRequestMessage create(
    final Actor rsvp, final int reqId,
    final message_type value)
{
    message_typeRequestMessage m =
        new message_typeRequestMessage(rsvp, reqId);
    m.value = value;
    return m;
}

```

Generates a create factory method for the request message type for a given message type.

```

translate_exp_member(reactor_member(message_type, message_id,
                                   reactor_number, block), actor_body)
=
private static class message_typeRequestMessage
    extends ajava.runtime.ActorRequestMessage
{
    public message_type value;
    public message_typeRequestMessage(final Actor rsvp, final int reqId)
    {
        super(rsvp, reqId);
    }
}

react (message_typeRequestMessage reqMessage) {
    message_type message_id = reqMessage.value;
    translate_reactor_block(block)
    create_condition_checks(actor_body)
}

declare_fork_actors(reactor_member)

```

Generates a request message class for a reactor, translates the reactor, and generates fork actor classes for every fork block in the reactor.

```

translate_exp_member(when_reactor_member, actor_body)
=
    translate_member(when_reactor_member, actor_body)

translate_exp_member(method_member(name, return_type,
                                   param_1, ..., param_n, block), actor_body)
=
    translate_member(method_member, actor_body)

translate_exp_member(event_member(actor_interface, event_id), actor_body)
=
    translate_member(event_member, actor_body)

```

```

translate_exp_member(field_member, actor_body)
=
    translate_member(field_member, actor_body)

translate_exp_member(constructor_member, actor_body)
=
    translate_member(constructor_member, actor_body)

translate_member(reactor_member(message_type, message_id,
                                reactor_number, block), actor_body)
=
    react (message_type, message_id)
        translate_stmt(block)

translate_member(when_reactor_member(condition, reactor_number, block),
                actor_body)

    protected class reactor_numberCondition {}

    translate_member(reactor_member(
        reactor_numberCondition, msg,
        reactor_number, block))

```

Translates a condition reactor into a message type (which is sent whenever the condition evaluates to true) and translates the underlying block as if it were a standard reactor.

```

create_condition_checks(actor_body(member_1, ..., member_n))
=
    create_condition_checks(member_1)
    ...
    create_condition_checks(member_n);

create_condition_checks(when_reactor_member(condition,
                                            reactor_number, block))
=
    if ( condition ) this <-- new reactor_numberCondition();

```

Generates a check to be placed at the end of every reaction, for a condition reaction, such that after every reaction has completed the condition is checked, and when it becomes true the correct message is delivered.

```

translate_member(method_member(name, return_type,
                               param_1, ..., param_n, block), actor_body)
=
    return_type name(param_1, ..., param_n)
        translate_stmt(block)

```

Translates an actor method member, translating all event assignments etc.

```

translate_member(event_member(actor_interface, event_id), actor_body)
=
    public final Event<actor_interface> event_id
        = new Event<actor_interface>();

```

Translates an event into a final instance of the event actor, which may be receive Subscribe and Unsubscribe messages, and relays any other messages it receives to all of its subscribers.

```

translate_member(field_member, actor_body) = field_member

```

```

translate_member(constructor_member, actor_body) = constructor_member

```

```
translate_reactor_block(reactor_member(message_type, message_id, block))
=
```

If count_forks(block) == 0 (i.e. no forks in the reactor) then:

```
    translate_stmt(block)
```

Else (i.e. at least one fork in the reactor) then:

```
    blockActor();
    message_typeForkActor frk = new message_TypeForkActor(
                                this, message_id);
    frk <-- new ajava.runtime.Continue();
```

Translates a reactor block, so that if it contains fork blocks, it is replaced with creating an instance of the local fork actor class, and sending it a continue message so that it starts executing the reaction. This is done so that all local variables are translated into fields of the containing fork actor, and are therefore accessible by nested forks.

```
count_forks(fork_statement) = 1
```

```
count_forks(return_statement) = 0
```

```
count_forks(break_statement) = 0
```

```
count_forks(expression_statement) = 0
```

```
count_forks(block(statement_1, ..., statement_n))
=
    count_forks(statement_1) +
    count_forks(statement_2) +
    ...
    count_forks(statement_n)
```

```
count_forks(if_statement(condition, true_statement, false_statement))
=
    count_forks(true_statement) +
    count_forks(false_statement)
```

```
count_forks(switch_statement(block_1, ..., block_n))
=
    count_forks(block_1) +
    ...
    count_forks(block_n)
```

Counts the number of fork blocks in the statement(s); doesn't include nested fork blocks.

```
translate_stmt(statement_block(statement_1, ..., statement_n))
=
{
    translate_stmt(statement_1)
```

```

        translate_stmt(statement_2)
        ...
        translate_stmt(statement_n)
    }

translate_stmt(if_statement(condition, true_stmt, false_stmt))
=
    if ( condition ) translate_stmt(true_stmt)
    else translate_stmt(false_stmt)

translate_stmt(switch_statement(exp, group_1, ..., group_n))
=
    switch ( exp ) {
        translate_stmt(group_1)
        ...
        translate_stmt(group_n)
    }

translate_stmt(switch_group(case_1, ..., case_n, stmt_1, ..., stmt_n))
=
    case_1
    ...
    case_n
    translate_stmt(stmt_1)
    ...
    translate_stmt(stmt_n)

translate_stmt(identifier_expression(id))
=
    If id == "this" then:
        THIS_LINK

    Translates all references to "this" in a fork actor's continuation to THIS_LINK so that it will refer to the
    containing actor, not the fork actor to which the task was delegated.

translate_stmt(assignment_expression(op, lhs, rhs))
=
    If lhs is instanceof Event then:
        If op is "+=" then:
            lhs <-- new Event.Subscribe(rhs);
        Else if op is "-=" then:
            lhs <-- new Event.Unsubscribe(rhs);

    Translates event subscriptions and unsubscriptions to sending subscribe, and unsubscribe messages.

translate_stmt(return_statement(return_value))
=
    {
        Response responseMessage = new Response(requestMessage);
        responseMessage.value = return_value;
        requestMessage.sendReply(responseMessage);
        return;
    }

    Translates a return statement into the sending of a response message to the requesting actor.

```

```

translate_stmt(fork_statement(fork_id, statement_1,..., statement_n, block))
=
    fork_id _fork_id = new fork_id(OWNER_RECEPTIONIST);
    translate_fork_stmt(statement_1, 1)
    translate_fork_stmt(statement_2, 2)
    ...
    translate_fork_stmt(statement_n, n)

```

Translates a fork statement into the creation of the relevant fork actor, and translates each of the fork statements (expression actor invocations executed concurrently) into a request message send.

```

translate_stmt(fork_statement(fork_id, statement_1,..., statement_n, block))
=
    fork_id _fork_id = new fork_id(OWNER_RECEPTIONIST);
    translate_fork_stmt(_fork_id, statement_1, 1)
    translate_fork_stmt(_fork_id, statement_2, 2)
    ...
    translate_fork_stmt(_fork_id, statement_n, n)

```

```

translate_fork_stmt(fork_id, local_variable_declaration(type, id,
    value_exp), num)
=
    translate_exp_actor_call(fork_id, value_exp)

```

```

translate_fork_stmt(fork_id, assignment_expression(lhs, rhs), num)
=
    translate_exp_actor_call(fork_id, rhs)

```

Translates an expression actor invocation into the sending of the correct request message.

```

declare_fork_actors(actor_class_body(member_1, ..., member_n), class_name)
=
    declare_fork_actors(member_1, class_name)
    ...
    declare_fork_actors(member_n, class_name)

```

```

declare_fork_actors(reactor_member(message_type, message_id, block),
    class_name)
=
    if count_forks(block) > 0 then:

aclass message_typeForkActor {
    class_name THIS_LINK;
    int MSG_WAITING_COUNT;
    int FORK_WAITING_COUNT;

    message_type message_id;
    declare_local_vars(block)
    declare_local_vars(statement_1)
    ...
    declare_local_vars(statement_n)

    public message_typeForkActor (
        final class_name THIS_LINK,
        message_type message_id) {
        this.THIS_LINK = THIS_LINK;
        this.message_id = message_id;
        this.MSG_WAITING_COUNT = count(statement_1, ..., statement_n);
        this.FORK_WAITING_COUNT = count_forks(block);
    }

    public react (ajava.runtime.Continue c) { continue(); }
    void continue() translate_fork_body(block)

```

```

public react (ajava.runtime.ForkDone d) {
    FORK_WAITING_COUNT--;
    if (FORK_WAITING_COUNT <= 0) done();
}
void done() { THIS_LINK <-- new ajava.runtime.UnblockActor(); }

declare_fork_actors(block, message_typeForkActor)
}

```

If a reactor contains one or more forks, generates a fork actor class that can be used to delegate the reaction, and which contains each local variable in its scope as a field of the actor. When it receives a Continue message it performs the reaction, and when all of its child forks have returned their ForkDone messages, it unblocks the owning actor, allowing it to receive buffered messages.

```

declare_fork_actors(block(statement_1, ..., statement_n), container_name)
=
    declare_fork_actors(statement_1, container_name)
    ...
    declare_fork_actors(statement_n, container_name)

declare_fork_actors(if_statement(condition,
                                true_stmt, false_stmt), container_name)
=
    declare_fork_actors(true_stmt, container_name)
    declare_fork_actors(false_stmt, container_name)

declare_fork_actors(switch_statement(block_1, ..., block_n), container_name)
=
    declare_fork_actors(block_1, container_name)
    ...
    declare_fork_actors(block_n, container_name)

declare_fork_actors(fork_statement(fork_id,
                                statement_1, ..., statement_n, block), container_name)
=
    aclass fork_idForkActor {
        container_name OWNER_RECEPTIONIST
        int MSG_WAITING_COUNT;
        int FORK_WAITING_COUNT;

        declare_local_vars(block)
        declare_local_vars(statement_1)
        ...
        declare_local_vars(statement_n)

        public fork_idForkActor(final container_name OWNER_RECEPTIONIST)
        {
            this.OWNER_RECEPTIONIST = OWNER_RECEPTIONIST;
            this.MSG_WAITING_COUNT = count(statement_1, ..., statement_n);
            this.FORK_WAITING_COUNT = count_forks(block)+1;
        }

        For every unique type of expression actor referenced in statement_1, ..., statement_n:
        declare_fork_reactor(expactor_type_1, statement, ..., statement)
        ...
        declare_fork_reactor(expactor_type_n, statement, ..., statement)

        void continue() {
            try {
                translate_fork_body(block)
            }
        }
    }

```

```

        } finally {
            this(new ForkDone());
        }
    }

    public react (ajava.runtime.ForkDone d) {
        FORK_WAITING_COUNT--;
        if (FORK_WAITING_COUNT <= 0) done();
    }
    void done() { OWNER_RECEPTIONIST <-- new ajava.runtime.ForkDone(); }

    declare_fork_actors(block, fork_idForkActor)
}

```

Generates a fork actor class to wait for the responses to the expression actor invocations in its fork statements, and when all have been received, executes the fork body and then returns a ForkDone message to its containing fork actor, to signal that it is complete.

```

declare_fork_reactor(expactor_type, statement_1, ..., statement_n)
=
    public react (expactor_type.Response responseMessage) {
        switch (responseMessage.getRequestID()) {
            translate_fork_stmt_reactor(statement_1, 1)
            ...
            translate_fork_stmt_reactor(statement_n, n)
        }
        if (MSG_WAITING_COUNT <= 0) continuation();
    }

```

Generates a reactor which responds to the response message for a particular expression actor type. The switch statements chooses which variable to assign the result to based on the request id, and when all of the responses have been received, executes the continuation (the fork body).

```

translate_fork_stmt_reactor(local_variable_declaration(type, id), num)
=
    case num: id = responseMessage.value;
    MSG_WAITING_COUNT--; break;

translate_fork_stmt_reactor(assignment_expression(lhs, rhs), num)
=
    case num: lhs = responseMessage.value;
    MSG_WAITING_COUNT--; break;

translate_fork_stmt_reactor(method_invocation_expression, num)
=
    case num: MSG_WAITING_COUNT--; break;

```

```

declare_local_vars(local_variable_declaration(type, id_1, ..., id_n))
=
    type id_1, id_2, ..., id_n ;

declare_local_vars(block(statement_1, ..., statement_n))
=
    declare_local_vars(statement_1)
    ...
    declare_local_vars(statement_n)

```



```

declare_local_vars(if_statement(condition, true_statement, false_statement))
=
    declare_local_vars(true_statement)
    declare_local_vars(false_statement)

```

```

declare_local_vars(switch_statement(block_1, ..., block_n))
=
    declare_local_vars(block_1)
    ...
    declare_local_vars(block_n)

```

Translates all local variables, into field declarations.

```

translate_fork_body(block(statement_1, ..., statement_n))
=
    {
        translate_fork_body(statement_1)
        ...
        translate_fork_body(statement_n)
    }

```

```

translate_fork_body(return_statement)
=
    translate_return(return_statement)

```

```

translate_fork_body(fork_statement(fork_id, statement_1, ..., statement_n,
                                   block))
=
    translate_stmt(fork_statement)

```

```

translate_fork_body(if_statement(condition, true_stmt, false_stmt))
=
    if ( condition ) translate_fork_body(true_stmt)
    else translate_fork_body(false_stmt)

```

```

translate_fork_body(switch_statement(exp, group_1, ..., group_n))
=
    switch ( exp ) {
        translate_fork_body(group_1)
        ...
        translate_fork_body(group_n)
    }

```

```

translate_fork_body(switch_group(case_1, ..., case_n, stmt_1, ..., stmt_n))
=
    case_1
    ...
    case_n
    translate_fork_body(stmt_1)
    ...
    translate_fork_body(stmt_n)

```

```
translate_fork_body(local_variable_declaration(type, id_1, ..., id_n,  
                                              val_1, ..., val_n))  
=  
  id_1 = val_1;  
  ...  
  id_n = val_n;
```

As all of the local variable declarations have been translated into field within a fork actor, the translates all local variable declaration initializers into assignments.

Appendix E.

This appendix lists code relevant to the implementation of the prototype translator. The definition of the base class used by all actors, and the actor program used to test the translator implementation are presented.

Actor Base Class

The following lists the abstract base class from which actors inherit via `ajava.lang.Actor`. This class describes the internal workings of a translated actor, and declares the private members used by translated actor classes.

```
package org.taj.ajava.runtime;

import org.taj.ajava.lang.*;

/**
 * The base class for all actors.
 * @author Tristan Aubrey-Jones
 */
public abstract class ActorBase implements Runnable {

    // count the actors in the system

    private static int ACTOR_COUNT = 0;
    private int ACTOR_NUM = ACTOR_COUNT++;

    public static int getActorCount() { return ACTOR_COUNT; }

    // constants

    /**
     * Defaults to this maximum number of reactions called
     * for each thread callback.
     */
    private static final int DEFAULT_MAX_REACTIONS_PER_CALLBACK
        = 5;

    // fields all actors have

    /**
     * This is the thread that reactions
     * run in. When a message is received
     * it buffers it, and either immediately
     * executes it (inside the calling thread)
     * or requests a call back from this thread.
     */
    private ActorExecutor executor;

    /**
     * When the actor requests thread time, this is
     * set to true, to that repeat requests arent made
     * while it is already waiting.
     */
    private boolean waitingForCallback;

    /**
     * Buffer for incoming messages. These are
     * handled when the thread calls the actor's
     * "callback" method.
     */
    private ActorMessageQueue incoming;

    /**
     * When true all incoming messages apart from
     * the UnblockActor message are buffered, and
     * not reacted to. This allows the actor to wait
     * for a reaction that that depending on the results
     * of actor calls, to complete.
     */
}
```

```

private boolean actorBlocked;

// constructor

public ActorBase() {
    // not waiting
    waitingForCallback = false;
    // not blocked
    actorBlocked = false;
    // create new message queue
    incoming = new ActorMessageQueue();
    // deliver initialize message
    deliver(new InitializeMessage());
    // get an executor
    executor = ActorExecutorManager.getExecutor();
    // request callback if messages on the queue
    // i.e. if receives the init message
    //if (incoming.count() > 0)
    //    requestCallback();
}

// methods

/**
 * If there is not already a request pending,
 * requests that the thread call the callback
 * method.
 */
private void requestCallback() {
    if (!waitingForCallback && executor != null) {
        executor.requestCallback(this);
    }
}

/**
 * Adds the given message to the message queue
 * and if the actor isnt blocked requests thread time.
 * @param msg
 */
protected void bufferMessage(ActorMessage msg) {
    // inc pending count
    ActorExecutorManager.incMessageCount();
    // buffer message
    incoming.enqueue(msg);
    // request callback
    if (!actorBlocked) requestCallback();
}

/**
 * Processes pending reactions.
 */
public void run() {
    // run with default max reactions
    run(DEFAULT_MAX_REACTIONS_PER_CALLBACK);
}

/**
 * Executes the reactions for up to "count"
 * messages in the incoming message queue, and
 * then if there are still unanswered messages,
 * it will request more time from the thread.
 */
public synchronized void run(int count) {
    // check if actor is now blocked
    if (actorBlocked) {
        // when unblocked will request
        waitingForCallback = false;
        return;
    }

    // process pending messages
    int lim = incoming.count() < count
        ? incoming.count() : count;
    while (lim > 0) {
        // get a message
        ActorMessage msg = incoming.dequeue();

```

```

        // process reaction
        processMessage(msg);
        ActorExecutorManager.decMessageCount();

        // check if actor is now blocked
        if (actorBlocked) {
            // when unblocked will request
            waitingForCallback = false;
return;

        }

        // decrement
        lim--;
    }

    // no longer waiting
    waitingForCallback = false;

    // request another callback if there are
    // messages remaining
    if (incoming.count() > 0) {
        requestCallback();
    }
}

// methods used for blocking an actor and buffer all
// incoming messages

/**
 * Blocks the current actor, so that it wont
 * process any messages (only buffer them) until
 * it is unblocked.
 */
protected void blockActor() {
    actorBlocked = true;
}

/**
 * Unblocks the actor and if there are
 * messages waiting requests thread time.
 */
protected void unblockActor() {
    actorBlocked = false;
    if (incoming.count() > 0) requestCallback();
}

// methods that need to be implemented by child classes

protected void deliver(InitializeMessage init) {
    // if an initializer is defined, this method will
    // be overridden to buffer the message with a valid
    // reactor id - thus if no initializer reactor is
    // define, no message is added.
}

/**
 * Processes the stalled message, by invoking the
 * correct reactor for it. Translator should generate
 * this method with a switch block for every reactor,
 * on the messages reactor id, typecasting the message
 * payload to the correct type, and invoking the correct
 * reactor method.
 * @param msg
 */
protected abstract void processMessage(ActorMessage msg); /* {
    // switch on the correct reactor id
    switch (msg.reactorId) {
        case 0: react_0((Integer)msg.payload); break;
        default: super.processMessage(msg);
    }
} */

// for each reactor

/*

// e.g. for reactor 0 (type Integer)

```

```

    public void deliver(Integer msg) {
        // buffer with correct reactor id
        bufferMessage(new ActorMessage(msg, 0));

        // or

        // process immediately
        react_0(msg);
    }

    private void react_0(Integer g) {

        // reactor code goes here

    }

    */
}

```

Translator Test Program

This test actor program was used to test the translator implementation, by ensuring that correctly behaving Java was emitted for the source that follows:

```

public actor Actor1 implements Entrypoint {

    // list iterator iterates over all
    // values in the list when receives run
    // to use extend from this and override
    // reaction to Object, which is called
    // for every iteration.
    public aclass Iterator {
        private java.util.List list;

        public Iterator(java.util.List list) {
            this.list = list;
        }

        public static class Run {}
        public react(Run r) {
            if (list.size() > 0) this(list.size()-1);
        }

        public react(Object v) {}

        private react(int it) {
            this(list.get(it));
            if (it > 0) this <-- it - 1;
        }
    }

    public static aclass Event {
        // current list of subscribers
        private java.util.List subscribers;

        public Event() {
            subscribers = new java.util.ArrayList();
        }

        // registers a subscriber
        public static class Subscribe {
            Actor a;
            public Subscribe(Actor a) {
                this.a = a;
            }
        }
        react (Subscribe s) {
            subscribers.add(s.a);
        }

        // sends to all subscribers
    }
}

```

```

private aclass SendAll extends Iterator {
    private Object msg;
    public SendAll(java.util.List list, Object msg) {
        super(list);
        this.msg = msg;
    }
    public react(Object target) {
        Actor.sendMessage(target, msg);
    }
}

// receives a message to send out
react (Object msg) {
    SendAll sndr = new SendAll(subscribers, msg);
    sndr <-- new Iterator.Run();
}

// Expression Actors

private actor Times2 returns int {
    public react(int i) { return i * 2; }
}

// times an integer by two
private aclass Doubler returns int {
    react(int x) {
        System.out.println("double called");
        return x*2;
    }
}

public aclass Actor2 {

    // request counter
    private int num = 0;

    public final Event RequestReceived = new Event();

    public final Doubler makeDouble = new Doubler();

    public react (int i) {
        // raise event
        RequestReceived <-- i;
        System.out.print("request number: ");
        System.out.println(num);

        // uses forks and actor requests
        i = Times2(i);
        System.out.println(i);
        int b = 2;
        b = Times2(i);
        System.out.println(b);

        // increments req number
        // (happens in final receptionist)
        num++;
    }

}

public react(String[] args) {
    System.out.println("Hello World!");
    Actor2 a2 = new Actor2();
    a2.RequestReceived <-- new Event.Subscribe(this);
    a2 <-- 2;
    a2 <-- 3;

    // test public final actors
    int i = a2.makeDouble(1);
    System.out.print("a2.makeDouble(1) returns ");
    System.out.println(i);
}

public react (int i) {
    System.out.print("a2.RequestReceived event occured: ");
    System.out.println(i);
}

```

```
}  
}
```


Appendix F.

This appendix lists the example actor programs written to evaluate the language, and the translator implementation. Examples were chosen to evaluate the usability of the language in different programming situations.

Dining Philosophers Example

This program implements a solution of the dining philosophers problem, to evaluate how good the language and translator is at solving common IPC problems.

```
public aclass Table {

    private Clock clk;

    private int numPlaces;
    private Fork[] forks;
    private Philosopher[] diners;
    private int[] states;

    public Table(Clock clk, int numPlaces) {
        this.clk = clk;
        this.numPlaces = numPlaces;
        this.forks = new Fork[numPlaces];
        this.diners = new Philosopher[numPlaces];
        this.states = new int[numPlaces];
        createPlace(numPlaces-1);
    }

    private void createPlace(int index) {
        forks[index] = new Fork(index);
        diners[index] = new Philosopher(this, index);
        clk.subscribe(diners[index]);
        states[index] = State.THINKING;
        if (index > 0) createPlace(index-1);
    }

    private int left(int id) {
        id--;
        if (id < 0) id += numPlaces;
        return id;
    }

    private int right(int id) {
        id++;
        if (id >= numPlaces) id -= numPlaces;
        return id;
    }

    // when a philosopher becomes hungry, updates its
    // state, and tries to eat
    public react (Philosopher.IsHungry msg) {
        // update state
        states[msg.id] = State.HUNGRY;

        // try and eat
        tryEat(msg.id);
    }

    // when receives some forks from a philosopher
    // records the fact it is thinking, and tries
    // serve its neighbours
    public react (MoveForks frks) {
        // return forks
        forks[frks.id] <-- frks.lhf;
        forks[right(frks.id)] <-- frks.rhf;
        states[frks.id] = State.THINKING;

        // try left and right
        tryEat(left(frks.id));
    }
}
```

```

        tryEat(right(frks.id));
    }

    // if the philosopher is hungry, and neither
    // neighbour is eating, then pass it its forks.
    private void tryEat(int id) {
        // if hungry, and left not eat, and right not eating
        if (states[id] == State.HUNGRY &&
            states[left(id)] != State.EATING &&
            states[right(id)] != State.EATING)
        {
            // move forks to the philosopher
            states[id] = State.EATING;
            diners[id] <-- new MoveForks(id,
                                      forks[id], forks[right(id)]);
        }
    }
}

```

```

import java.util.Random;

public class Philosopher {

    private int id;
    private Table table;
    private Fork lhs, rhs;
    private int state;

    public Philosopher(Table table, int id) {
        this.table = table;
        this.id = id;
        this.state = State.THINKING;
        randomDelay();
    }

    // begins thinking, moves forks back
    // to the table, and delays a random time
    private void think() {
        // give forks back
        table <-- new MoveForks(id, lhs, rhs);
        // remember: lhs and rhs are destructive reads
        // as are linear objects

        // change state
        state = State.THINKING;
        // think for random time
        randomDelay();
    }

    // becomes hungry, and announces to table
    // that it is hungry
    public static class IsHungry {
        public int id;
        public IsHungry(int id) { this.id = id; }
    }

    private void hungry() {
        state = State.HUNGRY;
        table <-- new IsHungry(id);
    }

    // receives forks simultaneously, and
    // begins to eat
    public react (MoveForks frks) {
        this.lhs <-- frks.lhs;
        this.rhs <-- frks.rhs;
        eat();
    }

    // eats for a random time
    private void eat() {
        state = State.EATING;
        randomDelay();
    }

    // uses clock pulses, and a random countdown

```

```

// to delay for a random period when thinking
// and eating.
private static final Random RNG = new Random();
private int countdown;
private void randomDelay() {
    countdown = RNG.nextInt(10) + 1;
}
public react (Clock.Tick t) {
    countdown--;
    if (countdown == 0) proceed();
}
private void proceed() {
    if (state == State.THINKING) {
        hungry();
    }
    else {
        if (state == State.EATING) {
            think();
        }
        else;
    }
}
}

```

```

public linear class Fork {
    private int id;

    public Fork(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}

```

Quick sort example

This example implements a simple parallel Quicksort, in both Java and the actor language to explore how well the language solves common divide and conquer parallel tasks.

Benchmarks

Array Size	1 Core			6 Core		
	Sequential	Actors	Threads	Sequential	Actors	Threads
1,000,000	547ms	554ms	564ms	972ms	612ms	693ms
100,000	52ms	51ms	54ms	77ms	67ms	73ms
10,000	3ms	6ms	21ms	7ms	30ms	37ms

Actor version ~ 30 lines

```

import org.taj.ajava.util.*;

/** Sorter, performs quicksort on integer arrays
    using array partitioning */
public aclass IntSorter returns IntegerArray {

    public react (IntegerArray array) {
        if (array.size() <= SorterMethods.MIN_PARTITION_SIZE) {
            // sorts manageable chunks sequentially
            SorterMethods.sortArray(array);
            return array;
        } else {

```

```

        // partition inplace around a pivot value
        int pivotIndex = SorterMethods.choosePivotIndex(array);
        int pivotNewIndex =
            SorterMethods.partitionArray(array, pivotIndex);

        // split the array in two (underlying array
        // remains unsplit, allows safe distribution
        // of array)
        int[] indices = new int[1];
        indices[0] = pivotNewIndex;
        IntegerArray[] parts = array.split(indices);

        // sort both halves concurrently
        IntSorter lhs = new IntSorter();
        IntSorter rhs = new IntSorter();
        fork ( parts[0] = lhs(parts[0]);
              parts[1] = rhs(parts[1]); )
        {
            // join the sorted halves back together
            // (they still exist inplace, so no
            // copying necessary)
            array.merge(parts);

            // return the sorted array
            return array;
        }
    }
}

```

Java version ~ 70 lines

```

import org.taj.java.util.*;

public class IntSorterThread {

    /**
     * Sorts an array using recursive quicksort.
     */
    private static class WorkerThread extends Thread {

        private IntegerArray array;

        public WorkerThread(IntegerArray array) {
            this.array = array;
        }

        public IntegerArray removeArray() {
            IntegerArray a = array;
            array = null;
            return a;
        }

        public void run() {
            if (array.size()
                <= SorterMethods.MIN_PARTITION_SIZE)
            {
                SorterMethods.sortArray(array);
            } else {
                // partition array
                int pivotIndex =
                    SorterMethods.choosePivotIndex(array);
                int pivotNewIndex =
                    SorterMethods.partitionArray(array, pivotIndex);

                // split into two halves
                int[] indices = new int[1];
                indices[0] = pivotNewIndex;
                IntegerArray[] parts = array.split(indices);

                // sort concurrently
                WorkerThread lhs = new WorkerThread(parts[0]);
                WorkerThread rhs = new WorkerThread(parts[1]);
                lhs.start();
                rhs.start();
            }
        }
    }
}

```

```

        // sorting...

        try {
            // wait for threads to finish
            lhs.join();
            rhs.join();
        } catch (InterruptedException ex) {
            throw (new RuntimeException(ex));
        }

        // join parts together
        parts[0] = lhs.removeArray();
        parts[1] = rhs.removeArray();
        array.merge(parts);
    }

}

/**
 * Inplace quicksort, delegated to threads.
 * @param array
 * @return
 */
public IntegerArray sort(IntegerArray array) {
    WorkerThread t = new WorkerThread(array);
    t.start();

    try {
        t.join();
    } catch (InterruptedException ex) {
        throw (new RuntimeException(ex));
    }

    return t.removeArray();
}
}

```

Event actor class

The following classes are used by the calculator example that follows to implement events, and demonstrate the ability to inherit protocols using actor inheritance. Here the list iterator could be used as a common base class for almost all iteration.

List Iterator

```

import java.util.List;

/** A list iterator iterates over all
 * values in the list when it receives run.
 * To use, extend from this and override
 * reaction to Object, which is called
 * for every iteration.
 */
public class ListIterator {
    private List list;

    public ListIterator(List list) {
        this.list = list;
    }

    public static class Run {}
    public react(Run r) {
        if (list.size() > 0) this(list.size()-1);
    }

    public react(Object v) {}

    private react(int it) {
        this(list.get(it));
        if (it > 0) this <-- it - 1;
    }
}

```

```
}  
}
```

Event Actor

```
import java.util.*;  
  
/**  
 * Event actor which relays all messages to  
 * it subscribers.  
 */  
public class Event {  
    // current list of subscribers  
    private List subscribers;  
  
    public Event() {  
        subscribers = new ArrayList();  
    }  
  
    // registers a subscriber  
    public static class Subscribe {  
        Actor a;  
        public Subscribe(Actor a) {  
            this.a = a;  
        }  
    }  
  
    public react (Subscribe s) {  
        subscribers.add(s.a);  
    }  
  
    // unsubscribe  
    public static class Unsubscribe {  
        Actor a;  
        public Unsubscribe(Actor a) {  
            this.a = a;  
        }  
    }  
  
    public react (Unsubscribe s) {  
        subscribers.remove(s.a);  
    }  
  
    // sends to all subscribers  
    private class SendAll extends ListIterator {  
        private Object msg;  
        public SendAll(List list, Object msg) {  
            super(list);  
            this.msg = msg;  
        }  
        public react(Object target) {  
            Actor.sendMessage(target, msg);  
        }  
    }  
  
    // receives a message to send out  
    react (Object msg) {  
        SendAll sndr = new SendAll(subscribers, msg);  
        sndr <-- new ListIterator.Run();  
    }  
}
```

Calculator example

This example demonstrates the language's ability to create modular interactive applications, making good use of event driven programming.

```
public actor Main implements Entrypoint {  
    private Calculator calculator  
        = new Calculator();  
}
```

```

    public react (String[] args) {
        // show calculator
        calculator <-- new AFrame.Show();
    }
}

```

```

import javax.swing.*;
import java.awt.*;

public class Calculator extends AFrame {

    // members
    final ALU alu;
    final NumberBox display;
    final ADigitPad digitPad;
    final AOpPad opPad;

    // constructor
    public Calculator() {
        super("Calculator");
        frame.setSize(300, 200);

        // alu
        alu = new ALU();

        // create panel
        APanel panel = new APanel();
        contentPane <-- new AContainer.AddComponent(panel);

        // number display
        display = new NumberBox();
        display.OnOperation <-- new Event.Subscribe(alu);
        alu.OnResult <-- new Event.Subscribe(display);
        panel <-- new AContainer.AddComponent(display);

        // create digit pad
        digitPad = new ADigitPad();
        digitPad.OnClick <-- new Event.Subscribe(display);
        panel <-- new AContainer.AddComponent(digitPad);

        // create operation buttons
        opPad = new AOpPad();
        opPad.OnClick <-- new Event.Subscribe(this);
        panel <-- new AContainer.AddComponent(opPad);

        // register for key presses
        OnKeyTyped <-- new Event.Subscribe(this);
        digitPad.OnKeyTyped <-- new Event.Subscribe(this);
        opPad.OnKeyTyped <-- new Event.Subscribe(this);
    }

    // receives an operation command
    public react (char op) {
        if (validOperator(op)) {
            display <-- new Operation(op);
        }
    }

    // key press
    public react (KeyEvent e) {
        // numeric digit
        if (Character.isDigit(e.character) || e.character == '.') {
            display <-- e.character;
        }
        // return
        else if (e.character == '\r' || e.character == '\n')
            this('=');
        // operator
        else this(e.character);
    }

    // operations
    public class Operation {

```

```

        public char operator; // operation to perform
        public double operand; // value of number register
        public Operation(char op) {
            this.operator = op;
        }
    }

    private static boolean validOperator(char c) {
        switch (c) {
            case '+': case '-':
            case '*': case '/':
            case '^': case '=':
                return true;
            default:
                return false;
        }
    }
}

```

```

public class ALU {

    private double register;
    private char operator = '=';

    public final Event OnResult = new Event();

    public ALU() {
        register = 0.0;
    }

    // perform operation
    public react (Calculator.Operation op) {
        System.out.print(op.operand);
        System.out.print(op.operator);

        // perform op
        switch (operator) {
            case '+': register += op.operand; break;
            case '-': register -= op.operand; break;
            case '*': register *= op.operand; break;
            case '/': register /= op.operand; break;
            case '^': register = Math.pow(register, op.operand);
                       break;
            // if was equals then new reg is operand
            case '=': register = op.operand; break;
        }
        operator = op.operator;

        // raise event
        System.out.println(" returns " + Double.toString(register));
        OnResult <-- register;
    }
}

```

```

import javax.swing.*;

public class NumberBox extends AComponent {

    protected JTextField textField;
    private boolean reset = false;
    private boolean donePoint = false;

    public final Event OnOperation = new Event();

    public NumberBox() {
        super(new JTextField(15));
        textField = (JTextField) (component);
        textField.setHorizontalAlignment(JTextField.RIGHT);
        textField.setText("0");
    }

    // appends a character (digit, or point)
    // to the current numeral
    public react (char c) {
        // digit
        if (Character.isDigit(c)) {

```



```

        if (reset) {
            textField.setText("" + c);
            reset = false;
        }
        else if (textField.getText().equals("0")) {
            if (c != '0') textField.setText("" + c);
        } else {
            textField.setText(textField.getText() + "" + c);
        }
    }
    // decimal point
    else if (c == '.' && !donePoint) {
        textField.setText(textField.getText() + '.');
        donePoint = true;
    }
}

// injects the current operand into the operation
// and forwards it to the ALU
public react (Calculator.Operation op) {
    // pass to ALU
    op.operand = Double.parseDouble(textField.getText());
    OnOperation <-- op;
}

// sets the value, resulting from an ALU
// operation
public react (double v) {
    textField.setText(Double.toString(v));
    reset = true;
    donePoint = false;
}
}

```

```

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ADigitPad extends AContainer {

    protected final GridLayout layout;
    protected final ACharButton[] buttons;

    public final Event OnClick = new Event();

    public ADigitPad() {
        // create panel
        super(new JPanel());
        layout = new GridLayout(4,3);
        container.setLayout(layout);
        buttons = new ACharButton[10];
        createButtons(7); // 7,8,9,4,5,6,1,2,3,0
    }

    private void createButtons(int value) {
        createButton(value);
        if (value == 3) createButton(0);
        else if (value % 3 == 0) createButtons(value-5);
        else createButtons(value+1);
    }

    private void createButton(int value) {
        buttons[value] =
            new ACharButton(Character.forDigit(value, 10));
        buttons[value].OnKeyTyped <--
            new Event.Subscribe(OnKeyTyped);
        buttons[value].OnClick <-- new Event.Subscribe(OnClick);
        this <-- new AddComponent(buttons[value]);
    }
}

```