A Cup of Joe

An inherently concurrent and actor-based programming language

Project Report

dpt104f15

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title:
A Cup of Joe

Theme:
Programming Technologies

Project Period:
Spring Semester 2015

Project Group:
dpt104f15

Participant(s):
Birgir Már Elíasson
Johannes Lindhart Borresen

Supervisor(s):
Bent Thomsen

Copies: 4

Page Numbers: 61

Date of Completion:
June 9, 2015

Abstract:

In this thesis we present *Joe*, an actor-based programming language with a Java-like syntax. Our primary contributions are Joe itself, and in particular its concept of *Protocols* to denote communication between Actors, as well as a means of verifying programs written in *Joe*. This is done by translating Joe Actors and Protocols to equivalent templates for consumption by the UPPAAL verification tool. We find that the basics of the approach are sound, paving the way for further studies. Finally, we implement a solution to a problem first presented in the authors' prespecialisation report ("Sorting and Synchronising", 2014), finding that equivalent programs written in Joe are both more concise, being easier to read and maintain.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the authors.

Preface

> Many people with whom the author has spoken have expressed the opinion that programming under such circumstances will be impossibly complicated and will never be worth while.
> *S. Gill on parallel programming, 1958*

This report is the master thesis of Birgir Már Elíasson and Johannes Lindhart Borresen, Software Engineering students working at the Department of Computer Science at Aalborg University, Denmark.

The project is a continuation of our 9th semester report, *Sorting and Synchronising - Evaluating concurrent and parallel implementations of Santa Claus and Quicksort in X10, Java, Clojure, and Erlang*. We build upon the findings of said report by developing a new programming language for the Java Virtual Machine (JVM), called `Joe`, designed with the goal of supplanting the existing concurrency mechanisms found in `Java`.

The project is inspired by a project proposal suggested by the authors, with guidance from their supervisor to realise the idea.

Bibliography and appendix are found in the closing part of the report. All program code has been uploaded to the digital project library along with a copy of this report.

For interested parties without access to the AAU project library, access to the online source code repository (found at `https://bitbucket.org/nattfari/dpt104f15-code`) may be granted upon request.

Birgir Már Eliasson	Johannes Lindhart Borresen
`belias10@student.aau.dk`	`jborre10@student.aau.dk`

Contents

1 Introduction **9**
1.1 Motivations 10

2 Analysis **13**
2.1 Previous Work 13
2.2 Related Work 14
2.3 Summary 21

3 Design **23**
3.1 Language design criteria 23
3.2 Features 24

4 Implementation **35**
4.1 Grammar 35
4.2 Translation 37
4.3 Verification 39
4.4 Runtime 42

5 Implementing Santa Claus **45**
5.1 One to one Implementation 45
5.2 Alternative Implementations 46

6 Evaluation **49**
6.1 Language Design 49
6.2 Implementation 50
6.3 Verification 50
6.4 Santa Claus Implementation 51

7 Conclusion **53**
7.1 Future Work 54

List of Listings **57**

Appendix **65**

A Appendix **69**
A.1 Grammars 69
A.2 Alternative Santa Claus Implementation 75

Chapter 1

Introduction

The MOS Technology 6502 was an affordable microchip used in many consumer-grade appliances, sometimes credited as the catalyst of the home computer revolution of the 1980's. In 1984, Rockwell International introduced a microcomputer that combined two 6502 cores on a single chip[13][26], perhaps making it the earliest example of a consumer-grade Symmetric Multiprocessing (SMP) system.

However, it was not until 2005, when both AMD[2] and Intel[22] introduced consumer-grade dual-core processors, that multi-processing came into the mainstream. This enabled entire industries of entertainment, research, big data, among others, to evolve.

Despite the leaps in processing capability and new applications, programming safe, concurrent or parallel applications remains difficult. Issues of parallel computing date back more than 50 years, from acknowledgement [19] to solutions [7].

According to the TIOBE index, the ten most popular programming languages are sequential by nature and mainly offer concurrent building-blocks through libraries that complement a threaded memory model. An example is `Java`, an object-oriented relative of `C`. `Java` first appeared in 1995 and had its first stable release in 1996, however it was not until version 5.0 in 2004 that measures were taken to improve on its broken memory model and supply utilities to deal with the difficulties of thread programming. At the time of writing, `Java` triumphs in popularity, followed by other sequential relatives of the `C` language, such as `C#`, `C++`, and `Objective-C`, which also incorporate threads.

Opposing these is `Erlang`, concurrent and distributed programming language with language primitives for these features. Founded on the actor model, it utilises message passing and offers language-level features for managing processes. Compared to the more popular `C`-like languages, however, `Erlang` is relegated to 36th place on the TIOBE index, following `Scala`, a functional, actor-based JVM language, on 32nd.

If parallelism is the future of computing, why is it that none of the most pop-

ular languages embed it at their core? More so, why are languages with this interest in mind not more widely used?

While analysing the reasons for this is beyond the scope of this project, we suggest that programming languages' staying power is a result of their platforms and age; porting entire systems between languages is costly and prone to errors, while languages with a very broad platform support are more appealing specialisation choice for professional programmers, granting more job opportunities. Thus we consider the issue from a different perspective: "If you can't beat them, join them." If it is not possible to supplant a programming language, would it instead be possible to augment it in a way as to afford better and safer parallelism? *This is the core of our thesis.*

We propose a new language, designed to be easy to learn by programmers familiar to Java-like languages, where concurrency is not just an option - it is built into the core of the language. Further, we wish to augment the language in such a way as to aid the programmer in avoiding some of the major pitfalls of concurrent programming:

The project itself builds upon the work and findings of the authors' prespecialisation, a report written as a precursor to this thesis. [6]. Referenced will be made throughout this report. We do, however, present an overview of that project's contents and results in Section 2.1.

1.1 Motivations

Working through various problems and their implementations in `X10`, `Java`, `Erlang`, and `Clojure`, we experienced the vast differences in expression across paradigms and coding philosophies. Languages exhibit very different properties in regards to synchronisation, non-termination and even globally accessible variables.

A key discovery, however, was the realisation that even today, programming languages in practice have very little, if anything, to remove the two key issues facing anyone programming concurrent software: Deadlocks and race conditions. Further, the languages surveyed had no clean way to implement Trono [37]'s Santa Claus problem as specified. While representations were possible given some allowances, it was more difficult to represent the properties of the system as part of the language's syntax or type system. Instead, solutions had to rely on conditionals and self-imposed structure to establish a protocol of communication.

We envision a language where concurrency is inherent to its very structure, implicit and safe in its use. Where many languages use libraries to introduce concurrency using existing formalisms (such as classes and functions) we instead want concurrency to have its own syntax or, even better, to underlie the entire code. Optimally, this language will complement an existing, mainstream language, allowing programmers to either convert complex, concurrent applications to a simpler programming model gradually, or implement concurrent parts

of a system in this new language, applying interoperability to keep sequential or trivial application parts in its original language.

Chapter 2

Analysis

2.1 Previous Work

During the fall of 2014, we analysed four programming languages within the context of two problems, one parallel and one concurrent. For each implementation, the languages and their solutions was evaluated based upon four criteria:

Complexity relates to code length, required effort when translating a model into code, and the difficulty of ensuring correctness of code.

Scaling speaks to the possibility of scaling an implementation, how the scaling would be performed, as well as an evaluation of the effort required to do so.

Maintenance considers the effort required to control technical debt and the solution's overall readability.

Performance is relevant only to parallel problems and examines the trade-off between time spent and efficiency gained when implementing a solution in a language.

The languages were chosen based on their TIOBE[8] ranking as well as the purity of their concurrency model. The latter refers to how deeply concurrency was built into the language as opposed to being added as a library (compare, for example, the `pthreads` library of `C` with the communication primitives of `Erlang`).

Java was chosen as a baseline language on which to compare the others. It is `C`-like, popular, and while most of its concurrency is featured through library support, it does contain some syntax specifically for the purpose.

Clojure is a `Lisp`-like, functional language that runs on the JVM. Beyond this, several of `Clojure`'s concurrency mechanisms rely on Software Transactional Memory (STM).

Erlang as an actor-based, functional language. Unlike Java or `Clojure`, `Erlang` contains keywords and syntax specifically for the handling of messages instead of using function call formalities.

X10 stands as a language built for high performance and scalability, right out of the box.

Two problems were chosen for implementation in all four languages:

The Santa Claus problem as described by Trono [37]. It models the problem of simulating Santa Claus as well as his elves and reindeer, primarily focusing on the handling of various issues of concurrency; deadlocks, race conditions, and semaphores.

Quicksort was used as a parallel problem to be solved in the four programming languages. Of particular interest was the ease of work distribution and scaling performance on each language.

Findings

`X10`, `Java` and `Erlang` all afforded relatively simple mappings for the Santa Claus problem, containing abstractions (in the form of classes or processes) to act as each party of the problem itself. `Clojure`, with its `Lisp`-like syntax presented a much greater challenge in writing a solution that did not diverge significantly from the source description. If this was disregarded, more maintainable versions could be achieved.

As Quicksort can be implemented with relatively few language constructions (namely, functions, lists and a parallelism mechanism), all languages displayed simple implementations. `X10`, `Java`, and `Clojure` displayed high performance even with large lists to sort, possibly as a result of all three languages running on the JVM. `Erlang`, on the other hand, exhibited very long running times compared to the other surveyed languages. While an `Erlang` solution with comparable performance exists, it is both verbose and complex.

Future Work

Based on the findings of the prespecialisation we hypothesised the possibility of designing a language with cleaner and more succinct concurrency mechanisms with an acceptable performance trade-off for the benefit of cleaner design and safer multi-processing.

2.2 Related Work

Ours is not the first attempt to bring improved concurrency mechanisms to the JVM or modern programming languages in general. Throughout this section we

present the primary models and concepts used in our work, with a final argumentation for the niche that we envision the language will fill.

2.2.1 Actor Model

The actor model is tailored for artificial intelligence and parallel execution, described by Hewitt [20], among others. In this view, actors are primitives in computation, encapsulating data and communicating via asynchronous message passing. This minimises the risk of deadlocks and data races, making the model a desirable option for implementing concurrent systems.[1] Actors are offered both as the primary computation model in languages and as a language extensions, in forms of libraries and frameworks.

Languages or Libraries

The programming language `Scala`, first released in 2003, applies actors and futures in its concurrency model. It runs on the JVM and is interoperable with `Java`. `Scala` actors are implemented as a library, allowing for a mix of concurrency models to co-exist. This also increases the risk of breaking the safety properties conferred by a pure actor model. The risk itself is not the problem. However, developers tend to fall back on known technologies when faced with difficult problems. For example, Tasharofi, Dinges, and Johnson [30] shows that Scala developers have a tendency to introduce threads into actor systems, claiming library- and actor model inadequacies or arguing efficiency. What ever the reason is, it is clear that language-level features are more desirable than a library. This would push developers into seeking solutions within the actor model, effectively guarding its safety properties.

Limitations and Opportunities

As explained in our prespecialisation project [6] there are some issues regarding synchronisation in the actor model. It may allow for a modular and simpler design of concurrent systems, however, correctness verification is not one of its properties. It is therefore in the hands of the programmer to verify the soundness of a solution's communication patterns. However, we consider the actor model to be an excellent medium for solving problems such as Trono's Santa Claus problem, where independently acting agents work within a single system to achieve the program's overarching goals.

2.2.2 Behavioural Types

Type checking is one of various formal methods for validating system correctness. As a key feature of compiler optimisation, types are often syntactically expressed in programming languages or inferred during compilation. Benjamin C.

Pierce provides this definition of a type system in his book *Types and Programming Languages*

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.[28]

A behavioural type system does not only classify phrases according to the kinds of values they compute, it also classifies the phrases according to *how* they compute.

Reasoning about concurrent execution in systems can be difficult. This often results in errors that may be hard to debug and sometimes flawed products that, in the eyes of the user, inexplicably break down and stall. Behavioural types enable developers to better reason about the execution of their concurrent applications. Moreover, it can be utilised in tool support for system validation, raising compile-time errors when an implementation would not act as a specification dictates.

The following story about the Therac-25 [39] underlines the importance of better verification mechanisms for concurrent code, particularly in safety-critical systems:

> Therac-25 was a radiation therapy machine produced by a Canadian nuclear science and technology laboratory. Due to a race-condition error, it could administer radiation doses thousands of times larger than normal, resulting in radiation poisoning, which in three cases led to death. The problem was introduced in software interlocks which replaced previously used hardware ones. Being safety-critical, these interlocks were vulnerable to an error which was nearly impossible to reproduce as it involved both timing and input variables.

Only through careful investigation did underlying causes of the issues surface, resulting in the creation of a development standard for medical device software[39].

In the following sections we outline a selected subset of behavioural type systems to analyse their suitability in our proposed programming language.

Type and Effect Systems

Nielson and Nielson [27] expounds on type and effect systems, an *inference-based* (as opposed to *flow-based*) technique of static analysis. Originally conceived by Lucassen and Gifford [25], the type and effect system's types are augmented with *effects* that reflect what happens with a variable of a certain type during program execution.

One application is Deterministic Parallel Java, an extension to the Java programming language that guarantees deterministic semantics [4]. In Deterministic Parallel Java (DPJ), programmers annotate the code in order to partition

the heap into *regions*, explicitly noting when methods read or write to different regions. This, in turn, allows the DPJ compiler to analyse program flow and ensure that the program exhibits deterministic behaviour under parallel conditions. This happens in much the same way as *checked exceptions* in `Java`, in which method declarations must explicitly state whether they internally handle exceptions or pass them up the call stack.

While the authors show DPJ to be sound, it does further increase the verbosity of Java in ways that could be considered redundant.

Session Types

Session types were originally used to describe linear interactions between two threads. In this sense, session types can be applied within the context of processes communicating via messages in order to verify their interactions according to some specification or protocol. [9]

A resulting session has notions for sending and receiving ends of channels, the types communicated on these channels, and the order in which they should be communicated. Sessions have been extended to include choice and recursion, enabling branching and infinite sessions.

Well-typed programs conforming to a binary session type system are safe in regards to two classes of programming errors, namely *communication-* and *race errors*. *Communication errors* comprise incompatible interaction. For example when a process tries to send a message to a receiving process which does not accept the message. If two processes both try to send or receive on the same channel at the same time, it would constitute a *race error*.

In an introductory paper to the subject, Vasconcelos, Aceto, et al. [38] presents a version of session types that allow for branching communication between more than two parties.

In the paper [21] session types are integrated into `Java` and presented in a full implementation. In this implementation sessions are declared as protocols inside classes and session type branching and session type recursion are explicit syntactical constructs. While the implementation promises correctness and low overhead, its verbosity is not appealing. The added syntax and injection of protocols into class declarations might intimidate novice developers.

Contracts

Usually described in programming languages with similarities to process algebrae or Labeled Transition System (LTS)s, contracts do not only describe possible actions, they directly describe actions that are allowed to be taken. Similar to session types, no communication errors will occur between two complementary contracts if they are compatible. Additionally, Fournet et al. [18] adopted CCS algebra for denoting contracts, contributing stuck-free conformance which roughly corresponds to deadlock exemption.

Hu, Yoshida, and Honda [21] use contracts, referring to them as sessions. While this might entail stuck-free conformance, a desired property, its verbosity remains an issue.

Contracts offer much in terms of safety and correctness. Coupled with the actor model, we consider this a powerful formalism for describing and ensuring correctness of actors as well as their ensuing communications.

2.2.3 Model Checking

As systems grow increasingly more complex, ad hoc methods of verification lose effectiveness, calling for formal testing methods to assure correctness of the programs. Traditionally developers have relied on debugging, peer reviews, and unit testing for verification of their work. Peer review is a good example of static analysis with reliable results. Empirical studies indicate that the technique, although often performed in an ad hoc manner, catches between 31-93% of bugs[5]. It can, however, be difficult to reason about concurrent and parallel systems making peer reviews less effective.

Model checking offers a greater capacity for verification, through brute force simulation while applying mathematically proven verification algorithms on a representation of the system.

Baier and Katoen [3] define model checking as follows:

> Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

Among the various tools created for the purpose of model checking, UPPAAL is a state-of-the-art automata verifier, complete with a description language, a simulator, and a graphical user interface. Released in 1995 it has been applied to a number of case studies and is actively maintained and extended.[31]

Verifying Actor Communication

As explained in Section 2.2.1, coordinating actor communication can be difficult, especially as the system grows in complexity. We suggest applying model checking as a means of verifying the communication patterns and specifications of actors.

D'Osualdo, Kochems, and Ong [15] propose a method to verify Erlang-style concurrency. The method entails deriving an abstract model of the system, called an Actor Communicating System (ACS), from control flow analysis of the program, which may be interpreted as a vector addition system (a type of LTS). Properties are verified using a tool called *Soter* which translates the model into a petri net and properties into queries, which in turn are verified by a coverability-checker called B_{FC}.

We envision using a similar technique in the verification of actors and their contracts to verify the protocols of communication.

2.2.4 JVM and Languages

The JVM is a portable platform with a large community [12]. Therefore, languages targeting the JVM may benefit from third-party community projects right out of the box. These languages typically bring different paradigms or coding styles compared to Java, allowing system designers to pick the right tool for the job while retaining interoperability with the entire JVM code base. In the following we present a list of programming languages that run on the JVM, in an attempt to find one that already achieves what we aim to do.

Jython is `Python`, ported to `Java`. It is a dynamically typed scripting language. It implements `Python`'s threading library while also supporting the concurrency packages of `Java`.[23]

Clojure is a `Lisp` dialect, thus predominantly a functional language. It focuses on STM for its concurrency.[11]

Scala presents itself as a pure object-oriented as well as functional language. It relies on `Java` for its concurrency support.[34]

Ozma is based on `Scala` and provides concurrency constructs similar to futures in a semi-transparent fashion. Strictly speaking, `Ozma` runs on the Mozart VM, but uses a modified `Scala` compiler tool-chain for most of its compilation.[14]

Groovy is described as an object-oriented programming language. Valid `Java` code is typically also valid Groovy code, albeit with dissimilar semantics. It primarily relies on `Java`'s concurrency constructs, although third-party libraries wraps this for convenience.[33]

Kotlin is a statically typed language that compiles to a variety of targets, including the JVM, of which `Scala` was an inspiration. It relies on interoperability with the Java libraries for concurrency.[24]

Ceylon is an imperative, statically typed and object-oriented language with syntax derived from C. It also relies on interoperability with the JVM for its concurrency.[10]

Fantom is a Java-like language and uses separate actors for its concurrency.[17]

Thorn isolates processes (*components*) and only allows transmittal via message passing.[35]

The majority of these languages expects the programmer to use the languages' interoperability features to invoke Java's own concurrency utilities, a few offer their own. However, they all expect the designer to allocate concurrency where needed, instead of building the language around it.

2.2.5 The E Programming Language

Only one language stood out during our search for languages with features and formalisms similar to those we propose in this report: The E programming language. E is an object-oriented, dynamically typed programming language designed by Mark S. Miller, and others from Communities.com, in 1997. While not inherently concurrent, it introduces a concurrency model based on messages, event loops and promises. The creators claim that their promise-pipeline, inspired by the Joule programming language, guarantees complete exemption from deadlocks and that E is essentially an actor system.

E is implemented on two platforms, Java and Common Lisp. In E, everything is an object and encapsulation is enforced by the language. The syntax is inspired by the C family of languages, primarily C and C, in an attempt make transition into the language easier on developers familiar with those languages.

In order to utilise the concurrent facilities of E special syntax must be used to return a promise instead of a value. The E documentation refers to this as a *send* action, hence the message concept. Any subsequent calls made on the returned promise must be sent correspondingly, unless an event construct is used. The event construct allows for safe direct method calls on promises using event triggers.

E boasts about its exemption from deadlocks, barring any human design oversights. For example, a concurrent system may stall, waiting on a message that is never sent. [29][16]

Summary

E presents the closest match to what we propose as an actor-based language with strong safety features within concurrency, while remaining easy to learn and use. However, E fails to meet our criteria for such a language. We conclude this section with an overview of the reasons for this.

Syntax
: While E draws inspiration from the C family languages (most notably Python)[16], introducing several syntactical constructs for concurrent features. We fear this departure from the core syntax of similar languages steepens the learning curve unnecessarily.

Concurrency model
: E, while based on the actor model, exposes an event loop to programmers, which may alienate some due to the added complexity. We suggest that an implicit message passing semantic would greatly benefit new programmers.

Safety
: The E creators boast about the language's security in context of encap-

> sulation and untrusted code. However, it lacks any facilities for the verification of communication between modules in a program, leaving it to the individual programmer to ensure correctness. Here, we propose using features of model checking to define specifications and verify their corresponding implementations.

While `E` displays many of the traits we seek in a programming language, it does not fit squarely within our criteria. Neither is it to be found on the TIOBE index [36], being virtually unknown within the context of popular languages.

Judging from its outdated documentation, missing source repositories, and the date of last change to its unofficial git repository, we assume it is no longer actively maintained.

2.3 Summary

Multiple attempts have been made to extend and enhance languages like `Java` with the introduction of various extensions and libraries. These include behavioural types, actor frameworks, and language-level features. While many of these attempts are sound, in the context of popular languages, we observe problems with their verbosity, introduction of superfluous new language features or complexity.

Instead, we intend to design a language that combines an inherently concurrent memory model with actors and contracts within the syntactic context of a popular programming language.

We call this language `Joe`.

Chapter 3

Design

A programming language's syntax and formalisms govern the methods by which a programmer will express a solution to a problem. Depending on the target domain, a language may feel more restrictive when compared to general-purpose languages. This should not be considered a limitation as much as an aid in correctness.

Consider a simple example: any C-like language against a strictly functional language such as `Erlang` or `Scheme/Lisp`. An imperative language typically gives the programmer great freedom of expression (global variables, protocols, state and paradigm mixing) while the stricter functional languages will generally discourage such features in order to foster a better program design.

In this chapter, we present `Joe`, an inherently concurrent actor-based programming language running on the JVM. In order, we present the criteria for the design of the language, based on findings from the pre-specialisation, leading to a high-level description of the language's most prominent features. We conclude with a rough overview of the language's architecture in regards to workflow.

3.1 Language design criteria

`Joe` stems from the desire to create a language that simplifies a programmer's task of creating concurrent or parallel applications on a wide spectrum of computer systems. As a consequence, `Joe` targets the JVM, allowing it to reach a very large install base of Personal Computers (PCs) and, potentially, smart phones.

It is designed in such a way as to easily avoid (or altogether eliminate) race conditions and deadlocks, while fostering a syntax that is intended to allow a more concise and maintainable translation of the Santa Claus to program code. Notably within the following key areas:

Concurrency is a foundation, on which the language is built on. Many languages (`C`, `Java`, `Clojure`) expose concurrency constructs either as library functions or function-like statements. For example, `C` only allows for con-

currency (and parallelism) through OS-dependant libraries like `pthreads`, while `Java`'s concurrency comes through specific class libraries. Even `Erlang` does not *initiate* concurrency through language syntax, rather through function calls in the runtime (`spawn/1 thru spawn/3`). `Joe` intends to make concurrency intrinsic to the language and how actors operate.

Syntax recognisability governs how easily new programmers will familiarise themselves with the syntax and its corresponding semantics. Realising that many of the most popular programming languages are part of the `C` family[36], we imagine message passing represented in the guise of method invocation between objects. The call-return expectation fits well with a synchronised, binary protocol.

Safety should be guaranteed to a high degree by the language or its tools (such as the compiler toolchain). By introducing concepts known in other contexts such as interfaces and protocols, programmers will be able to apply the new features of `Joe` without needing to learn a completely new theory of computation.

3.2 Features

There are many facets to `Joe`, covering aspects of syntax and semantics as well as approaches to the verification mechanisms. Throughout this section, we will present major features of the language.

`Joe`'s two primary defining features, when compared to `Java`, is the introduction of *Actors* and *Protocols*. Actors take the place of classes in `Java` as the primary form of structure. They are said to *understand* one or more Protocols, replacing `Java`'s interfaces. In this manner, they declare the messages they understand, and the message sequence for the communication protocol. s

3.2.1 Syntax

`Joe`'s syntax is derived from `Java`, and it is our aim that programmers familiar with `C` and its derivatives will be able to easily accustom themselves to the language. Most principally, this comes in the form of defining *Protocols*, *Actors* and their behavioural logic. While the style will be recognisable, core keywords and the semantics of code will have drastic differences. On the face of it, a Java developer should be able to write a valid `Joe` program without much coaching, making use of the concurrency features almost to a fault. Compare a basic Java class in Listing 1 `Joe` Actor in Listing 2.

Note the very close similarities between declarations of class and Actor as well as method and message endpoint definitions. Their styles are purposely kept syntactically close. Their meaning, however, are drastically different. Where

Listing 1 A simple Java class.

```
// In JavaInterface.class
interface JavaInterface {
  public void JavaMessage(Object sender);
}

// In JavaClass.class
class JavaClass implements JavaInterface {
  private String name;

  public JavaClass(String newname) {
    this.name = newname;
    System.out.println(this.name + " is in an initial state!");
  }

  @Override
  public void JavaMessage(Object sender) {
    System.out.println("JavaMessage was called!");
  }
}
```

Java methods may have a number of visibilities and can thus be called from outside the object, Joe Actors' fields are private, their only external interface being the message endpoints declared in an understood Protocol.

3.2.2 Actors

Actors are defined very much like classes in Java, see Listing 3

Message endpoints (or *message handlers*) are defined in a similar manner to methods in Java, with a name, list of formal parameters, and method body. An Actor may not define a handler that is not defined as part of its corresponding Protocol's input messages. There is a single special case as an exception to this rule: Constructors. An Actor may contain a constructor, setting initial values of fields and initiating communication as specified by its protocol.

Within a message handler's body, most expressions that are valid Java expressions remain valid Joe expressions. Variable declarations, assignment, and loops look like their Java counterpart. Sending a message to another Actor is syntactically equivalent to calling a method in Java. Semantically, however, an Actor will not wait for a response or the conclusion of its target's computations. It simply continues running its own statements. That is, sending messages is a non-blocking action.

Joe does add a notation not found within the method bodies of Java: Asynchronous *method* calls. By default, all messages to other Actors are sent asynchronously, while all calls to regular Java code is handled sequentially. To override this behaviour, any call may be prefixed with the @ sign. This wraps the calls

Listing 2 The JavaClass translated to Joe.

```
// In JoeProtocol.spec
protocol JoeProtocol {
  JoeProtocol() -> JoeMessage
  JoeMessage() -> end, JoeMessage
}

// In JoeActor.joe
actor JoeActor {
  String name

  JoeActor(String newname) {
    name = newname
    print("I'm in an initial behavioural state!")
  }
}
```

Listing 3 An example of an Elf from the Santa Claus problem described in terms of a Joe Actor.

```
    actor Elf understands ElfProtocol {
      Santa santa

      Elf(Santa s) {
        santa = s
      }

      ProblemFixed() {
        // Problem was fixed. React to it!
      }
    }
```

return value (if any) in an implicit future, to be resolved when the return value is needed. Joe's runtime will try to re-order statements, running independent code as far as it can before blocking in anticipation of return values.

3.2.3 Protocols

Protocols are Joe's form of contracts between Actors. In order for an Actor to define any message endpoints, it must first declare that it supports (or *understands*) a specific Protocol. Protocols are defined similar to Java interfaces, with annotations regarding understood messages. See Listing 4.

Each line reads as follows: First a *message definition*, followed by its formal parameters, allowing additional data to be passed as part of a message. Directly proceeding this declaration, a number of possible *protocol steps* can be given. The list of steps denotes which messages are valid immediately following the

Listing 4 A basic Protocol.

```
1    protocol MessageProtocol {
2      MessageProtocol() -> ^SendMessage
3      SendMessage(String message) -> ReceiveMessage
4      ReceiveMessage(String response) -> ^SendOtherMessage
5      SendOtherMessage(Boolean confirmation) -> end
6    }
```

one given in the message definition - the list itself is not a strict ordering of messages. Each step must correspond to a message definition, or the special step `end` which signifies the final step of a protocol. A step prefixed with ˆ denotes an *outgoing* message while an unadorned step is for an *incoming* message. A Protocol is binary, written from the perspective of one half of a session of communication. An Actor may declare themselves to understand the second party role of a Protocol by negating it with ˆ in its `understands` list of understood protocols. An example is shown in Listing 5.

Listing 5 An Actor, declaring that it understands `MessageProtocol` as the second party.

```
1    actor Recipient understands ^MessageProtocol {
2      SendMessage(String msg) {
3        sender.ReceiveMessage(msg)
4      }
5      SendOtherMessage(Boolean confirmation) {
6        print("Received confirmation " + confirmation + ".")
7      }
8    }
```

All Protocols must define an initial “pseudo message” that declares all legal first steps in the protocol. This message must carry the same name as the Protocol itself.

Thus, Listing 4 can be read as follows: The Protocol `MessageProtocol` is defined by the message definitions `SendMessage` (which transfers a `String` value), `ReceiveMessage` (also a `String`), and `SendOtherMessage` (which transfers a boolean value). An Actor that acts as first party to `MessageProtocol` must first output a `SendMessage`, then receive a `ReceiveMessage`, finally sending a `SendOtherMessage`, terminating the protocol session.

3.2.4 Polymorphism

While `Joe` does not support inheritance in its current form, it does allow for polymorphism along the same lines as `Java`. An Actor `SomeActor` that understands `ProtocolA` and `^ProtocolB` can be assigned to a variable of any of the three types. Note that while `SomeActor` may be assigned to a variable of type `ProtocolA` it may

not be assigned to a variable of type `^ProtocolA`. A Protocol and its negated counterpart are considered two mutually exclusive types.

Listing 6 An Actor that understands ProtocolA and the negation of ProtocolB.

```
1  protocol ProtocolA {
2    ProtocolA() -> ^Hello
3    Hello() -> end
4  }
5
6  protocol ProtocolB {
7    ProtocolB() -> Goodbye
8    Goodbye() -> End
9  }
10
11 actor SomeActor understands ProtocolA, ^ProtocolB {
12   // Both protocols only contain output messages.
13 }
```

Listing 7 An Actor that tries to assign a SomeActor instance to various fields.

```
1  actor OtherActor {
2    ProtocolA actorA
3    ProtocolB actorB
4    SomeActor anActor
5
6    OtherActor(SomeActor myActor) {
7      // Valid
8      acActor = myActor
9      // Valid
10     actorA = myActor
11     // Invalid. SomeActor understands negated ProtocolB
12     actorB = myActor
13   }
14 }
```

This feature allows an Actor to declare interactions with an Actor of a very specific type, or any Actor that adheres to a specific Protocol. See Listing 7 for an Actor that tries to assign a SomeActor (Listing 6) to its fields.

3.2.5 Message Passing

In designing `Joe`, two approaches to message passing were considered: Active Erlang-style explicit sending and receiving as opposed to a more passive event-style format. Here we present both along with a short discussion, before concluding with a final choice.

In both variants, message passing is masked as method invocation on other

actors (`targetProcess.message()`), becoming an asynchronous invocation. The discussion erupts when considering how to describe message reception in `Joe`.

Erlang style

In Erlang, messages in the queue are handled explicitly. A message is sent using the `!` operator (such as `targetProcess ! { message }`) and received in the other end with the use of the `receive` block, see Listing 8.

Listing 8 An Erlang `receive` block.

```
1    receive
2    { message } -> DoSomething()
3    end
```

Messages are filtered via pattern matching, allowing messages to be postponed or even disregarded as the process runs.

In `Joe`, this style would be represented by explicit `receive` blocks akin to those of Erlang, with the added static correctness of Java. As such, a receive block cannot attempt to receive messages that are undefined in the accepted protocols and their current state. Listing 9 gives an example of how this might look.

This style has the following benefits:

Explicit blocking through the use of `receive` blocks offers programmers a high degree of flow control.

Methods defined on the actor level can be leveraged for program structure, encapsulating actor behaviour from external code.

Validation can offer finer granularity in results, given the level of annotation resulting from explicit receives.

Event style

The event style focuses on passive message endpoints rather than explicit `receive` blocks, each endpoint (a message *handler*) containing the code necessary to handle any incoming messages. Endpoints must match the protocols accepted by the actor. See Listing 10 for an example.

Compared to Erlang, event-style message passing has the following properties:

Methods cannot be used directly for structuring complex internal behaviour. This must be nested in a constructor and/or looping idle code.

Blocking is not possible. While this requires the programmer to think differently about a solution, it fosters more concise message handlers that rely on internal actor state for complex behaviour without too much handling code.

Listing 9 An example of what message receive would look like in `Joe`.

```
protocol PingPong {
  PingPong() => ^Ping
  Ping() => Pong
  Pong() => Ping
}

actor Pinger understands PingPong {
  // Single, repeating behaviour.
  Pinger(^PingPong other) {
    other.Ping()
    receive PingPong.Pong as msg {
      System.out.println("Received pong response.")
    }
  }
}

actor Ponger understands ^PingPong {
  // Single, repeating behaviour.
  Ponger(PingPong other) {
    receive PingPong.Ping as msg {
      System.out.println("Received ping request.")
    }
    other.Pong()
  }
}
```

Validation cannot be performed on single actors. As an actor will trivially accept any message for which its current protocol states allow (once it has completed any current message handling action), it must be put in unison with a whole (at least one opposing actor) in order to verify its interactions.

Decision

Both message passing styles have their own merits but are mutually exclusive (the roles of "method" definitions in the actor body have vastly different uses). We opt to implement the event style, focusing on the brevity of code and implicit handling of messages for programming ease.

3.2.6 Safety

In order to verify Actors' conformance to their own protocols as well as their communication patterns with other Actors, `Joe` leverages the model checking capabilities of UPPAAL. This requires that an Actor and its Protocols be translated to a format the UPPAAL can understand.

Listing 10 An example of `Joe` using event-style message passing.

```
// Protocol PingPong defined elsewhere

actor Pinger understands PingPong {
  // This acts as a constructor/initial code.
  Pinger(^PingPong other) {
    other.Ping()
  }

  Pong() {
    System.out.println("Recived pong.")
    sender.Ping()
  }
}

actor Ponger understands ^PingPong {
  // Constructor unnecessary, comes from Pinger

  Ping() {
    System.out.println("Recived ping.")
    sender.Pong()
  }
}
```

Protocol to LTS

Translation of a Protocol is quite simple. All message definitions are defined as locations in an LTS. The initial Protocol's state is also modelled as a location. Protocol steps from each become transitions from the message definition to the location named by the step. The transition synchronises on a channel bearing the target location's name. The special message `end` is a transition to a special "end" location, with an extra transition to the initial state, allowing the Protocol to "reboot". Listing 11 shows a Protocol in `Joe` while Figure 3.1 shows the resulting UPPAAL template. The negated protocol is created in the same way, reversing the meaning of all input and output messages.

Listing 11 A QA Protocol. One of two questions, containing a String, is sent. In response, either a boolean or String response is sent back. At this point the Protocol session ends.

```
protocol QuestionProtocol {
  QuestionProtocol() -> ^StringQuestion, ^BooleanQuestion
  StringQuestion(String question) -> StringAnswer
  BooleanQuestion(String question) -> BooleanAnswer
  StringAnswer(String answer) -> end
  BooleanAnswer(boolean answer) -> end
}
```

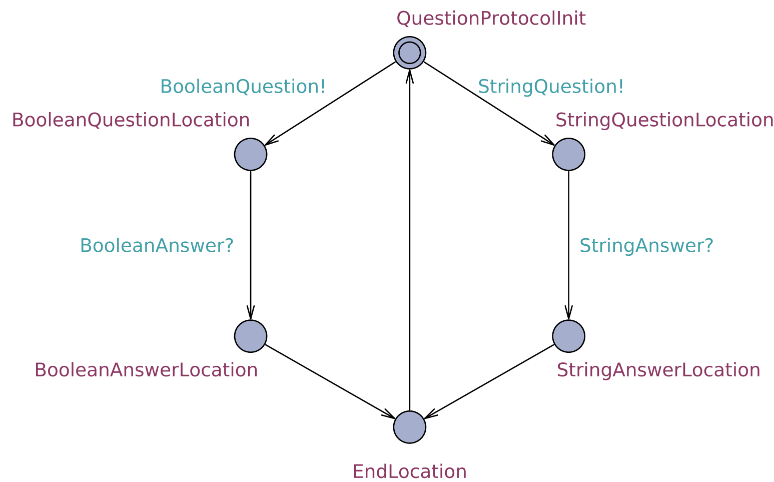

Figure 3.1: The resulting QuestionProtocol LTS in UPPAAL.

Actor to LTS

While Protocols are afforded a simple translation, Actors require a deeper analysis to mirror. While an Actor clearly declares its supported Protocols, and thus which channels it will use to synchronise on, the order of the synchronisations (particularly outgoing) is more complex to model correctly.

The core issue of the analysis lies in a construction of a Control Flow Graph (CFG) that is sufficient to describe the Actor's actions throughout its life time is not fully solved. In particular, loop constructs muddle the count of certain actions such as Actor instantiation and message sending, forcing the resulting UPPAAL model to remain imprecise in several key areas.

The premise of a translation goes as follows: set an initial location. Transition to a location denoting the beginning of the Actor's constructor (if one is present). Traverse the CFG of the constructor, creating a location for each node and a transition between each node. Whenever a message is sent, add a synchronisation to the matching transition matching the name of the message as was the case with a Protocol's translation. Once the constructor finishes, transition to a "hub" state. From the hub state, create transitions to locations matching all input messages that the Actor can receive. For each such location, map out the matching handler's method body in the same way as for the constructor. Once the CFG has been traversed, add a transition back to the hub location. Listing 12 shows an Actor definition while Figure 3.2 shows its corresponding UPPAAL template.

Verification

In order to verify that an Actor will adhere to the rules set forth by its supported Protocols, an UPPAAL system is constructed in which the Actor is instantiated

Listing 12 An Actor that understands the QuestionProtocol.

```
actor ModelActor understands QuestionProtocol {
  ModelActor(^QuestionProtocol answerer) {
    answerer.StringQuestion(
      "What is the answer to life, the universe, and everything?")
  }

  StringAnswer(String answer) {
    print("Got an answer:")
    print(answer)
  }

  BooleanAnswer(boolean answer) {
    if (answer) {
      print("The answer was yes.")
    } else {
      print("The answer was no.")
    }
  }
}
```

together with instances of the *negated* versions of its understood Protocol. The idea is to simulate the Actor's behaviour when combined with constructed counterparts.

Two queries are tested on the final system: `A[] not deadlock` to determine whether a circular system deadlocks at any point. If it does, `E<> deadlock and ActorName.ActorNameHub` to ensure that the Actor will only deadlock when receiving messages, not during message handling. An Actor may deadlock in the hub location if, for example, it does not send any new messages that would result in responses from any of its supported Protocols.

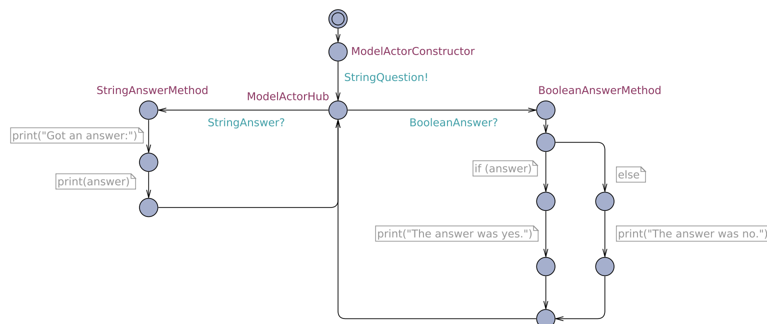

Figure 3.2: The UPPAAL LTS of the ModelActor Actor.

Chapter 4

Implementation

We have implemented a subset of the `Joe` language, focusing on features necessary to implement a working minimal solution to the Santa Claus problem.

`Joe` has been implemented using the Xtext framework. This has the benefit of streamlining several phases of language development, from grammar definition to code generation and runtime, allowing us to focus on the key aspects of the language rather than rudimentary language development tasks.

4.1 Grammar

`Joe` is split up into two connected grammars: One for the specification of Protocols, and one for Actors themselves. They can be found in Appendix A.1.

As `Joe` is intended to run on the JVM, downwards interoperability is an appealing prospect. The Xtext framework contains several grammars and modules, allowing the use of all or some parts it. Its largest module, Xbase, is a statically typed expression language that can be embedded as part of other languages. Using XBase requires the use of a model inferrer, rather than a code generator, to compile code. The inferrer maps elements of the source language to matching concepts in Java, and an XBase-specific generator translates the models to Java.

Using XBase as the basis for `Joe` would allow direct interoperability with Java, and indeed describing `Joe` in terms of XBase's grammar rules is quite simple. Translating `Joe`'s model to Java, however, is a complex undertaking. Some concepts are cleanly modelled such as Actors to classes, and message handler declarations to methods declarations. However, several of `Joe`'s essential features require more work than was feasible during this project. In particular, message *send* actions could not be implemented. In principle, a basic translation simply requires wrapping a method call to the target Actor in a future instantiation and invocation. Unfortunately, this would require disassembling XBase's own block expressions in order to retrieve the specific parts involved in method invocation. Similarly, this could principally be performed on the Abstract Syntax

Tree (AST)-like model used for code generation by overriding XBase's code generation behaviour. This, in turn, also proved too complex for the project.

In lieu of using XBase, a subset of `Joe` was implemented from scratch only using the `JavaVMTypes` grammar of Xtext (one of XBase's dependencies) along with elements of an example expression language bundled with Antlr (the parser used by XBase), SimpleExpressions. The grammar allows referencing Java types, opening the door for simpler interoperability (type usage and method invocation).

The following elements of `Joe` were implemented:

- Protocols in long form. A short-form Protocol format (inspired by CCS) was considered, but not implemented.
- Actor declarations. The only types recognised are those defined in `Joe` (Actors and Protocols) as well as list constructs.
- Partial message endpoint definitions. The expression language used in the body of message handlers is sufficient to express a concurrent system of interacting Actors, but has no Java interoperability, thus missing large features sets (such as file I/O, process handling, network connectivity, and more)

4.1.1 Body logic

The issue of using XBase as a base grammar is most obvious when defining the grammar for the bodies of message handlers. Using XBase, it could be defined in Xtext's grammar as follows:

```
MessageHandler:
  'def' name=ID '(' (parameters+=FullJvmFormalParameter
              (',' parameters+=FullJvmFormalParameter)* )?
         ')'
    body=XBlockExpression
```

The accompanying model inferrer would handle `MessageHandler` rules by creating a method with the given name, formal parameters, and pass it the body cleanly. This will generate a syntactically correct Java method in the containing class with accompanying scope and logic. However, it does so with the semantics of the Xtend language, not `Joe`. In particular, one core element is missing: asynchronous message passing and invocation. There are two distinct challenges in implementing the two variants.

For global function and Java interop invocations, the use of the "@" prefix is unique in its context within Java. While the character is used to denote *annotations* for a number of *declarations*, it is unused in the context of method invocations, making it safe for use within the XBase Expression language. Thus, the

problem becomes one of creating a rule that wraps invocations that are prefixed with "@", allowing the model inferrer to give them special treatment. However, the XBase grammar is quite expansive and, given its expressiveness, difficult to modify without affecting its entirety.

For asynchronous message sending, the problem is even greater. As the syntax of this feature matches that of method invocations perfectly (an intentional feature), the grammar is insufficient to determine whether a `Joe` element or Java element has been referenced, an issue of ambiguity. Thus, the inference of reference must come at a later step, exasperating the issue. It is necessary to override the model inferrer's behaviour when translating the correct XBase rules for method invocation, correctly identifying `Joe` messages (and translating them) or deferring to the original implementation.

Unfortunately, it was not possible to implement this functionality. In the interest of testing the verification step, a much simpler hand-made expression language was constructed. The Santa Claus problem was used as base case for implementation, guiding the minimal set of features necessary for an implementation and test of the verification step. This encompasses:

- Conditional branching (to allow Santa to continue in an idle state when insufficient elves or reindeer have contacted him).
- Looping (to allow Santa to send individual message back to each recipient).
- Dynamic arrays (to store references to elves and reindeer).
- A "print" statement (for visual debugging).
- Actor initialisation
- Actor-level fields.
- Assignment.

4.2 Translation

By design, `Joe` programs look almost like Java programs. This is not only a benefit when learning the language, but also when translating it to Java for further compilation.

Protocols

Translating Protocols to Java is a basic process of converting it to a Java interface. Listing 13 shows the original `ElfProtocol` implementation while Listing 14 shows the corresponding Java implementations. The workflow runs as follows:

For a given Protocol, create two interfaces, one normal and one inverted (in the example, one for `ElfProtocol` and one for `^ElfProtocol`. Declare methods for all the output messages in the normal Protocol's interface, and all inputs in the inverted Protocol's interface. The interface methods have the same signature as their Protocol counterparts, with one extra parameter - a reference to the sending Actor, via interface. A message,

```
Problem(String description, int ticketId)
```

in the protocol `MyProtocol` would be translated to

```
Problem(MyProtocolInverted sender, String description, int ticketId)
```

Listing 13 Protocol for the Elf.

```
1  protocol ElfProtocol {
2    ElfProtocol() -> ^Problem
3    Problem() -> Solution
4    Solution() -> ^Problem
5  }
```

Listing 14 `Java` equivalents of the Elf Protocol.

```
1  interface ElfProtocol {
2    public void Problem(ElfProtocolInverted sender);
3  }
4
5  interface ElfProtocolInverted {
6    public void Solution(ElfProtocol sender);
7  }
```

Using interfaces in the final Java program is not useful for the `Joe` programmer, but is a useful tool for toolchain and run-time developers, ensuring resulting code is correct.

Actors

Similar to Protocols, Actors are translated to Java classes with a few modifications. An Elf Actor in Listing 15 would be translated as shown in Listing 16. Most code is translated as-is, with the following exceptions:

print statements are translated to calls to System.out.println.

Message calls to other Actors are wrapped as callables and passed to `Joe`'s runtime.

Listing 15 A `Joe` implementation of an Elf

```
1   actor Elf understands ElfProtocol {
2     Santa santa
3
4     Elf(Santa s) {
5       santa = s
6       santa.Problem()
7     }
8
9     Solution() {
10      santa.Problem()
11    }
12  }
```

4.3 Verification

Verifying `Joe` code involves translating the program code of Actors and protocols into suitable representations within the UPPAAL program. In this project, this is achieved by constructing a CFG of the complete `Joe` program and translating it to a fitting abstraction that can be read and analysed by UPPAAL. In this section we describe the general rules used for CFG construction as well as the translation to UPPAAL XML files.

While a production version of `Joe` would include a complete toolchain, generating and validating XML files in the background, this was out of scope for the project. While invoking the UPPAAL processes from within a compile task is relatively simple, issues of proper file placement as well as distribution of the UPPAAL binaries, remain open questions for such an undertaking.

4.3.1 CFG

Construction of the CFG is attained via an object model traversal using the visitor pattern. When an Actor's CFG is requested, it traverses first the list of message end points, then their bodies, building up a set of locations and transitions as described in Section 3.2.6. CFGs are constructed no deeper than at the Actor level. That is, message passing and function invocations are not followed in the traversal.

4.3.2 XML construction

The `Template` class in the `uppaal` package functions as a serializer and in-memory representation of the Extensible Markup Language (XML) to be generated. Once an Actor's CFG has been generated within it, matching XML can be output as a string. `Template` outputs *partials*, or incomplete UPPAAL files, as it only generates the single template relevant for the represented actor.

Listing 16 The Java implementation of an Elf

```
public class Elf extends JoeLangActor implements ElfProtocol {
  Santa santa;

  Elf me = this;

  public Elf(Santa santa) {
this.santa = santa;
System.out.println("Elf lives. Problem to Santa");
Runtime.submitTask(santa, new Callable<Void>() {
      @Override
      public Void call() throws Exception {
        santa.Problem(me);
        return null;
      }
});
  }

  @Override
  public void Solution(ElfProtocolInverted sender) {
System.out.println("Received a solution. Got a new problem!");
// TODO Auto-generated method stub
Runtime.submitTask(santa, new Callable<Void>() {
      @Override
      public Void call() throws Exception {
        System.out.println("ElfActor->SantaActor runnable");
        santa.Problem(me);
        return null;
      }
});
  }
}
```

A full UPPAAL XML file is comprised of three distinct sections: global declarations, template definitions, and template instantiation. While the template definition section can be filled procedurally, declarations and instantiations cannot be filled until the entire system has been parsed.

Declarations

The UPPAAL XML file's declarations section contains declarations of the message definitions used throughout the program, in the form of channels. In order to avoid name clashes, the entire program must be parsed before writing to the file.

Template instantiation

A key element to the verification step is the instantiation of a sufficient number of UPPAAL templates. Consider a simplified `Santa` implementation, shown in Listing 17

Listing 17 Santa Claus, defined in JoeLang

```
actor Santa understands ^ElfProtocol {
  ElfProtocol[] elves
  Problem() {
    elves.add(sender);
    if (elves.size() >= 3) {
      for (elf : elves) {
        elf.Solution()
      }
    }
  }
}
```

It is clear that if no more than two `ElfProtocol` actors send a `Problem` message to the actor, it will not proceed with the next step of the protocol for any of the instances, deadlocking. Now consider the matching program entry point for this example, in Listing 18.

Listing 18 Entry point for the Santa/Elf example.

```
actor Main {
  Santa santa
  Main() {
    santa = new Santa()
    fornum(1..3) {
      new Elf(santa)
    }
  }
}
```

A simple traversal of the Actor model would find exactly one occurrence of the constructor for the elf Actor. Assuming one instance of the Elf template would be enough may be tempting, but would obviously deadlock the system. In this case, however, counting the number of constructor calls is possible, given the static numbered loop, running exactly three times (iterators 1, 2, and 3). Consider, however, a more generic example is shown in Listing 19.

Now, the number of loops is unknown until run-time, too late for the verification step. In the short term, a brute force technique, adding a template instance and re-running the simulation, could alleviate small or trivial programs, but will prove ineffective after very little program growth. Currently, we know of no means for UPPAAL to answer queries with something other than a truth

Listing 19 Another entry point, this time using unknowable loop sizes.

```
actor Main {
  Santa santa;
  Main(String[] args) {
    santa = new Santa()
    for (arg : args) {
      new Elf(santa)
    }
  }
}
```

value. For example "which configuration of template instances are required for the system not to deadlock?" is not a valid query.

4.4 Runtime

The runtime guarantees Actor-local consistency by only allowing a single message to be handled for an Actor at a time. Many Actors may be active in this way, but an Actor will never concurrently handle two incoming messages. This is achieved by maintaining a list of messages in a global message queue, along with information regarding the thread, if any, that is currently running in the Actor.

Messages are implemented as `Java` Futures. As Actors are translated into regular `Java` classes, such classes from other libraries can transparently be considered Actors as well, treating method calls as messages. This way, their functionality would be preserved and their application within `Joe` would be consistent to that of Actors.

The runtime itself is implemented as a class of methods and private fields, all static. Actors, when compiled to Java, invoke these methods to submit tasks (wrapped messages) to the global message queue. A thread pool is used to concurrently run several message handlers (on separate Actors) simultaneously. Each message is tagged to ensure that only *one* thread operates on a given object at a time.

4.4.1 Queue

The queue encapsulated by the runtime is a `BlockingSkipQueue`, custom class heavily inspired by the `LinkedBlockingQueue`, a class in `Java`'s standard library, which is implemented as a linked list. However, messages are not handled in a strictly First In, First Out (FIFO) manner. As several, consecutive messages on a single Actor may await handling, free threads must search deeper in the queue for available messsages. Thus the runtime requires a custom queue implementation.

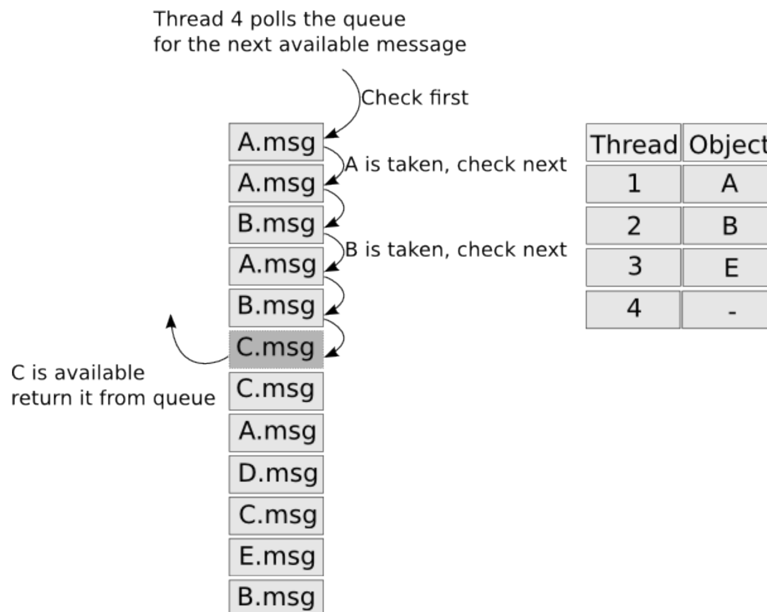

Figure 4.1: The BlockingSkipQueue being polled for the next available message

The `BlockingSkipQueue` utilises double locking to increase throughput, it grows dynamically, and it maintains a hash map over object-to-thread associations. When polled, the queue will search for the next available object according to the hash map, comparing each message tag to the hash map. If no threads are mapped to the object tag, or the tag matches the polling thread, the message is removed and returned from the queue. Figure 4.1 visually represents this process. Upon returning a message the queue updates the hash map accordingly before allowing access to other threads, see Figure 4.2. If the queue reaches its end, a null value is returned, effectively putting the polling thread in a wait mode, to not waste execution time until a new message is put into the queue.

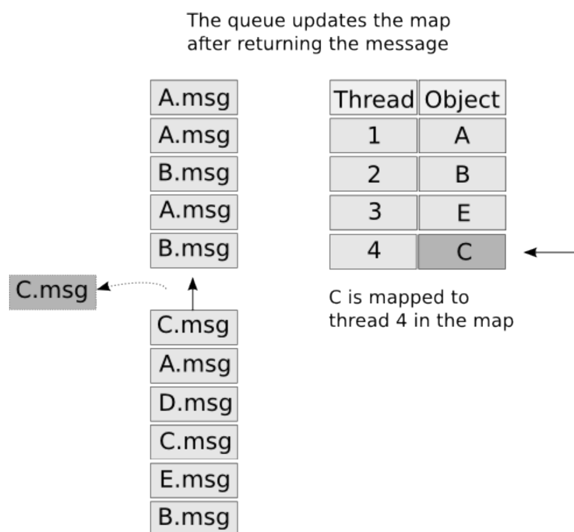

Figure 4.2: BlockingSkipQueue updating the hash map after returning a message

Chapter 5

Implementing Santa Claus

In this chapter, we present an implementation of the Santa Claus problem in `Joe`. It is designed with a simple "one-to-one" Protocol design, containing single Actors for each agent type in Santa Claus (Santa Claus, Elf, and Reindeer) and simple, circular Protocols for their communications. This is to demonstrate how `Joe` affords a much more concise implementation that remains easy to ready and understand. This is in accordance with the original design goals of the language.

5.1 One to one Implementation

Listing 20 describes the Protocols used between Santa and either an Elf or a Reindeer. Note they are simple and circular, essentially a call-response format.

Listing 20 The two Protocols used in the reference implementation of Santa Claus.

```
1 // In ElfProtocol.spec
2 protocol ElfProtocol {
3   ElfProtocol() => ^Problem
4   Problem() => Solution
5   Solution() => ElfProtocol
6 }
```

```
1 // In ReindeerProtocol.spec
2 protocol ReindeerProtocol {
3   ReindeerProtocol() => ^Arrive
4   Arrived() => Delivered
5   Delivered() => ReindeerProtocol
6 }
```

Listing 21 shows the `Joe` implementations of the Elf and Reindeer Actor. They act much like their Protocols would suggest - they respond with the countering part of the Protocol's steps. The two have been slightly differently implemented, although their behaviour in the context of the program is equivalent. `Elf` stores

a local reference to `Santa` which is used throughout its lifetime, while `Reindeer` instead references `Santa` once directly, only responding directly to the message sender of `Deliver` messages (which *is* `Santa`).

Listing 21 Implementations of Elf and Reindeer in `Joe`

```
// Elf.joe
actor Elf understands ElfProtocol {
  Santa santa

  Elf(Santa s) {
    santa = s
    santa.Problem()
  }

  Solution() {
    print("Santa has presented a solution. New problem!")
    santa.Problem()
  }
}
```

```
// Reindeer.joe
actor Reindeer understands ReindeerProtocol {
  Reindeer(Santa s) {
    s.Home()
  }

  Deliver() {
    print("Delivering presents with Santa. Coming home!")
    sender.Home()
  }
}
```

Finally, Listing 22 describes the Santa Actor. The implementation shows how one might implement one-to-many communications using the binary Protocols currently supported by `Joe`. For each message received from an `Elf` or `Reindeer` , `Santa` stores a reference to the sending Actor in an appropriate list. Once the count of list elements passes a threshold, `Santa` loops over the contents and sends the appropriate response message to each in turn.

5.2 Alternative Implementations

As discussed in the prespecialisation [6], several approaches to implemening Santa Claus were considered, depending on target language and supported constructs. Appendix A.2 contains a `Joe` implementation using queues as suggested in the `Erlang` implementation described in [6].

Listing 22 Joe implementation of Santa Claus

```
// Santa.joe
actor Santa understands ^ElfProtocol, ^ReindeerProtocol {
  Elf[] elves
  Reindeer[] reindeer

  Problem() {
    print("An elf has a problem")
    elves.add(sender)
    if (elves.size() >= 3) {
      print("Solving problems.")
      for (elf : elves) {
        elf.Solution()
      }
      elves.clear()
    } else {
      print("Not enough elves for a solution.")
    }
  }

  Home() {
    print("Reindeer returned home")
    reindeer.add(sender)
    if (reindeer.size() >= 9) {
      print("Delivering presents.")
      for (r : reindeer) {
        r.Deliver()
      }
      reindeer.clear()
    } else {
      print("Not enough to deliver, yet.")
    }
  }
}
```

Chapter 6

Evaluation

With the current iteration of Joe we believe we have accomplished most of what was intended in regards to language design. Joe's implementation in Xtext as well as the runtime remain in a prototype stage. While the verification model was sufficient for Santa Claus and several test cases, several limiations were identified and remain.

6.1 Language Design

One of the major contributions of this project is Joe's design itself. While the syntax was derived directly from Java, the underlying semantics and interactions are what make Joe unique. We believe the use of well-known concepts such as classes and interfaces, in the guise of Actors and Protocols, serve as excellent metaphors to flatten the learning curve while introducing powerful new features to the language.

Further, we believe the underlying notion, that all actors are concurrent, will aid programmers rationalise about them as independently acting agents in an environment, rather than a collection of objects with a single thread of logic shared between them.

Protocol Complexity

Joe's Protocols are binary as they support two Actors communicating with each other. More complex Protocols, with annotations for multiple parties and even constraints (to control message looping, for example), were considered, but did not reach a sufficient level of maturity for inclusion in the language.

Message Handling

There remains a contention in regards to perform message handling in Joe, as described in Section 3.2.5. Both styles, event and Erlang, present strong arguments in their own favour. We opted for the event style in the interest of min-

imalism and conciseness, but concede that the explicit blocking of `receive` expressions grants the programmer a useful layer of control and may contribute significantly as metadata for the verification step.

Inheritance

A notably missing feature from `Joe` is a stronger notion of inheritance. The polymorphic features, of combining and referencing Protocols into Actors, offers a high degree of expressiveness and flexibility in program designs. We envision a form of compositional Protocols, matching more closely with the behaivoural types described in the source material. We should like to see these improved Protocols together with actual Actor inheritance, once we can establish models that describe their interactions.

6.2 Implementation

The current implementation of `Joe` is sufficient to demonstrate core features within the context of a solution to the Santa Claus problem. Due to issues in regards to the malleability of Xtext's XBase expression language, the implemented grammar is not as expressive as that of its siblings or `Java` itself. We stand by Xtext as the tool of choice for implementing the language. Compared to many alternatives (Yacc, Lex, and derivatives), Xtext contains an entire toolchain, from tokenisation to code generation. It allows language developers to selectively modify the process, affording a high degree of control.

6.3 Verification

Initial tests of the verification model were promising, and correctly accepted some `Joe` implementations of Santa Claus, reporting errors when parts of the implementation were missing. During testing, however, we discovered several cases where the current verification model is insufficient.

The current CFG construction contains insufficient information to accurately reflect the many possible code paths of a `Joe` actor. We believe it is possible to make some changes to the language, and the CFG construction, to cover many cases. This was, however, not further examined. A particular issue is the resolution of "for each" loops which can, at best, be estimated.

UPPAAL Issues

Initial runs of verifications of `Joe` programs are promising, but two core issues remain. One is the greater challenge of constructing a CFG that accurately reflects active actors in any given state, in order to provide UPPAAL with sufficient information to decide the safety properties of the matching system.

Language	LOC
Joe (1:1)	~70
Joe (semaphores)	~83
Clojure	~85
Erlang	~102
Java	~183
X10	~256

Table 6.1: The approximate line counts of implementing Santa Claus in `Joe` and the four languages from [6] (`Java`, `X10`, `Erlang`, and `Clojure`). The lines of code are counted from the source while retaining its original coding style.

Secondly (and to our knowledge), UPPAAL does not support a query format that returns data beyond an acceptance or rejection of its formulae. It would greatly benefit diagnostics and program design, if queries could be enhanced to supply deeper responses to queries such as "under which conditions will the system fail or succeed?".

6.4 Santa Claus Implementation

The `Joe` implementation of Santa Claus can be written using less than 70 lines of code using one-to-one Protocols and about 83 lines of code using semaphore-like proxies (Appendix A.2). Compare this to the four languages surveyed in [6], shown in Table 6.1. Although `Joe`'s implementation is much shorter than its comparisons, note that coding style and metadata affects the representations as well. Most profoundly, perhaps, is the inclusion of packages in `X10` and `Java`, bloating the line count. If `Joe` was to be properly integrated into the JVM ecosystem, it may require the introduction of a similar notation, adding to its code size.

Applying the same four criteria from the prespecialisation [6], we find the following:

Complexity is a core focus of `Joe`'s design. While not specifically built to implement a Santa Claus solution, it was designed around the concept of actors and their interactions while retaining the algorithmic, imperative style of its heritage. Problems that can be described in terms of pseudocode algorithms or as a system of actors should be equally simple to implement.

Scaling of `Joe` lies within the domain of the implementation of the language and its run-time, not the individual programs. The current iteration of the runtime scales the number of threads with the available logical threads of the host Central Processing Unit (CPU), allowing several Actors to run in parallel.

Maintenance in `Joe` is a controlled activity. A code base could easily be refactored thanks to the strict checks of the protocols, while the encapsulation of data eliminates risks of data corruption on smaller changes. We do, however, defer largely to the maintenance evaluations of `Java` and `X10` in reference to `Joe`'s syntax heritage.

Performance can, unfortunately, not be determined at this time. We anticipate a trade-off in `Joe`, between low complexity and high performance. Further work is necessary to optimise the message passing mechanisms and threading models used to run the actors concurrently.

Chapter 7

Conclusion

In this, the final of our projects, we set out to design a new programming language. Perhaps not the most novel of prospects, as numerous languages are introduced every year. Maybe even futile, as very few of those languages reach mainstream appeal. We believe, however, that designing a new language such as ours has been a fruitful endeavour.

Motivated by the findings of our prespecialisation[6], the language `Joe` was designed to fit well within the ecosystem of an existing popular programming language (`Java`) with a deeply integrated concurrency model. Basing it on the actor model, we have augmented it with the notion of *protocols*, inspired by behavioural type theory and session types. This allows a programmer to define the communication between actors in terms of statically typed messages and their ordering. To our knowledge, this is a first within the context of JVM languages.

Finally, we implemented a prototype compiler toolchain using the Xtext framework[32] and `Java` futures, capable of compiling `Joe` elements to corresponding `Java` and UPPAAL[31] files. Using this prototype, we wrote and verified a solution to the Santa Claus problem[37] in `Joe`.

Joe

The programming language itself was designed to look very similar to `Java`. Actors bear many similarities to `Java` classes, and Protocols have close ties to `Java` interfaces. Indeed, program code within an Actor's message handlers is no different to that of `Java` method bodies. Two different syntaxes for the handling of incoming messages were considered. One was inspired by the `receive` blocks of `Erlang`, the other by that of event handlers found in many programming languages. Although both had their merits, the latter was chosen.

Verification

We added a mechanism in `Joe` that ensures program correctness via the verification of Actors when put in the context of their Protocols. This was accom-

plished by mapping actors and their corresponding models into equivalent LTSs in the UPPAAL verification tool[31], testing the ensuing system's safety properties. While the current model verifies Santa Claus, several improvements are necessary in regards to the translation of an Actor to an equivalent UPPAAL template.

Implementation

At this point, our implementation is sufficient to describe a solution to the Santa Claus concurrency problem that set our work into motion. Using the Xtext framework, we wrote two separate grammars, one each for actors and protocols. When compiled, Protocols are translated into `Java` interfaces while Actors are translated to `Java` classes implementing these interfaces. The generated classes make use of a separate `Joe` runtime with facilities for isolated execution of several Actors in parallel.

Santa Claus in Joe

In order to compare `Joe` with the programming languages surveyed in our prespecialisation (`Erlang`, `Java`, `X10`, and `Clojure`), we write a solution to Trono's Santa Claus problem[37], evaluating it using the same criteria: Complexity, Scalability, Maintenance, and Performance. We find that `Joe` affords a much shorter implementation (measured in lines of code), while retaining high maintainability and low complexity. Further, the `Joe` execution model is designed in such a way as to allow several Actors to run in parallel. It was not possible to measure performance, but we surmise `Joe`'s performance to be comparable (although slightly lower) to that of `Java`.

7.1 Future Work

The current iteration of `Joe` is a good indicator of what we envision as a modern programming language for the near future. Even so, much work remains before `Joe` reaches a stable state fit for production use. Here, we conclude the report with a number of suggestions for further study, not just within the context of `Joe`, but model checking as well.

Language Design

- There is a dichotomy in regards to the syntax of receiving messages in `Joe` . A further analysis of the two syntaxes may result in findings to find in favour of one over the over in regards to expressiveness, performance or issues with translation.
- A shorthand Protocol syntax was considered, equivalent to the current syntax and reminiscent of Calculus of Communicating Systems (CCS). If

equivalence could be proven, introduction of it in the language's grammar should be simple.

- Current Protocols only support one-to-one communication. Adding one-to-many Protocols, constraints (bounds for loops, for example) and the ability to compound or interweave Protocols is an appealing prospect.

Runtime

- The message queue implemented in the prototype build suffers considerable overhead, owing to the need for traversing the message queue from the top every time a thread is available for work. A different approach, perhaps with a different data structure or segmentation, may yield greater performance.

- While the current runtime is able to actively use all logical threads of a host CPU, it is not suitable for distributed computing. We suggest a runtime model where messages as well as Actors themselves can be transferred between runtime instances running on different machines. Going even further, optimisation strategies may involve considering where to place an Actor in order to ensure spatial locality in regards to its most common message targets.

Model Checking

- In a perfect world, any program's CFG could be accurately determined statically. This is not so. For example, bounds checking on loops in a CFG is in many cases not possible. While such a breakthrough may not be immediately forthcoming, we do believe it is possible to further improve upon the translation of Actors to UPPAAL models in order to improve the validation, not just in strength, but in granularity. Running a series of targeted queries may yield important validation errors to the programmer.

- The current models make a simple verification query, testing for a deadlock state in the resulting UPPAAL system. The result is boolean (either the query is satisfied, or it isn't). Working on the project, we found no means of retrieving more complex query results from UPPAAL. For example, it would be quite useful if UPPAAL could answer queries regarding valid ranges of values for loops in program code, to more concisely test with fitting numbers of template instantiations.

- It may be prudent to model the message queues between Actors to achieve a greater accuracy in the UPPAAL simulations. Whether this will improve results or simply slow down the verification step remains to be seen.

List of Listings

1 A simple Java class. 25
2 The JavaClass translated to Joe. 26
3 An example of an Elf from the Santa Claus problem described in terms of a Joe Actor. 26
4 A basic Protocol. 27
5 An Actor, declaring that it understands MessageProtocol as the second party. 27
6 An Actor that understands ProtocolA and the negation of ProtocolB. 28
7 An Actor that tries to assign a SomeActor instance to various fields. 28
8 An Erlang receive block. 29
9 An example of what message receive would look like in Joe. 30
10 An example of Joe using event-style message passing. 31
11 A QA Protocol. One of two questions, containing a String, is sent. In response, either a boolean or String response is sent back. At this point the Protocol session ends. 31
12 An Actor that understands the QuestionProtocol. 33
13 Protocol for the Elf. 38
14 Java equivalents of the Elf Protocol. 38
15 A Joe implementation of an Elf 39
16 The Java implementation of an Elf 40
17 Santa Claus, defined in JoeLang 41
18 Entry point for the Santa/Elf example. 41
19 Another entry point, this time using unknowable loop sizes. 42
20 The two Protocols used in the reference implementation of Santa Claus. 45
21 Implementations of Elf and Reindeer in Joe 46
22 Joe implementation of Santa Claus 47
23 The Protocols used in a queued Santa Claus implemented in Joe. . . 76
24 The Actors of Elf and Reindeer. Like their one-to-one counterparts, they are quite simple. 76
25 The queued implementation of Santa. Note how this Actor does not note what kind of queue is messaging it, only responding along the established Protocol steps. 77

26 The two queue implementations bear the brunt of the logic of this implementation. Individually, they mimic the one-to-one version of `Santa`'s lists of Actors to respond to. 78

List of Tables

6.1 The approximate line counts of implementing Santa Claus in Joe and the four languages from [6] (Java, X10, Erlang, and Clojure). The lines of code are counted from the source while retaining its original coding style. 51

List of Figures

3.1 The resulting QuestionProtocol LTS in UPPAAL. 32
3.2 The UPPAAL LTS of the ModelActor Actor. 33

4.1 The BlockingSkipQueue being polled for the next available message 43
4.2 BlockingSkipQueue updating the hash map after returning a message 44

Bibliography

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986. ISBN: 0-262-01092-5.

[2] *AMD Athlon™ 64 X2 Dual-Core Processor Product Data Sheet*. URL: `http://support.amd.com/TechDocs/33425.pdf` (visited on 05/19/2015).

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[4] Robert L Bocchino Jr et al. "A type and effect system for deterministic parallel Java". In: *ACM Sigplan Notices* 44.10 (2009), pp. 97–116.

[5] Barry Boehm and Victor R. Basili. "Software Defect Reduction Top 10 List". In: *Computer* 34.1 (Jan. 2001), pp. 135–137. ISSN: 0018-9162. DOI: `10.1109/2.962984`. URL: `http://dx.doi.org/10.1109/2.962984`.

[6] Johannes Lindhart Borresen and Birgir Mar Eliasson. *Sorting and Synchronizing. Evaluating concurrent and parallel implementations of Santa Claus and QUicksort in X10, Java, Clojure and Erlang*. Student Report. Aalborg University, 2014.

[7] E Brauoe. *"An Algorithm for The Detection of System Deadlocksâ€™*. Tech. rep. IBM Technical Report: TROO-791â€™IBM Data Systems Divisionâ€™Poughkeepsieâ€™NY, 1961.

[8] Tiobe Software BV. *Tiobe Index for January 2015*. URL: `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html` (visited on 01/09/2015).

[9] Giuseppe Castagna et al. "Foundations of Session Types". In: *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. PPDP '09. Coimbra, Portugal: ACM, 2009, pp. 219–230. ISBN: 978-1-60558-568-0. DOI: `10.1145/1599410.1599437`. URL: `http://doi.acm.org/10.1145/1599410.1599437`.

[10] *Ceylon: Welcome to Ceylon*. 2015. URL: `http://ceylon-lang.org/` (visited on 06/07/2015).

[11] *Clojure homepage*. 2015. URL: `http://clojure.org/` (visited on 06/07/2015).

[12] Oracle Corporation. *Community*. URL: `http://www.oracle.com/technetwork/java/community/index.html` (visited on 06/07/2015).

[13] *Datasheet: Rockwell International R65C00.* URL: http://www.datasheetarchive.com/dl/Scans-055/DSAIH000103824.pdf (visited on 05/19/2015).

[14] Sébastien Doeraene. "Ozma: Extending scala with oz concurrency". In: *Université Catholique de Louvain, Belgium, Masterarbeit* (2011).

[15] Emanuele D'Osualdo, Jonathan Kochems, and C.-H.Luke Ong. "Automatic Verification of Erlang-Style Concurrency". English. In: *Static Analysis.* Ed. by Francesco Logozzo and Manuel FÃ¤hndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 454–476. ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_24. URL: http://dx.doi.org/10.1007/978-3-642-38856-9_24.

[16] erights.org. *E's History.* URL: http://erights.org/history/index.html (visited on 06/06/2015).

[17] *Fantom homepage.* 2015. URL: http://fantom.org/ (visited on 06/07/2015).

[18] CÃ©dric Fournet et al. "Stuck-Free Conformance". English. In: *Computer Aided Verification.* Ed. by Rajeev Alur and DoronA. Peled. Vol. 3114. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 242–254. ISBN: 978-3-540-22342-9. DOI: 10.1007/978-3-540-27813-9_19. URL: http://dx.doi.org/10.1007/978-3-540-27813-9_19.

[19] Stanley Gill. "Parallel programming". In: *The Computer Journal* 1.1 (1958), pp. 2–10.

[20] Carl Hewitt. *What is Computation? Actor Model versus Turing's Model.* Also published as part of http://www.worldscientific.com/worldscibooks/10.1142/8306. URL: http://what-is-computation.carlhewitt.info/ (visited on 10/15/2014).

[21] Raymond Hu, Nobuko Yoshida, and Kohei Honda. "Session-Based Distributed Programming in Java". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming.* ECOOP '08. Paphos, Cypress: Springer-Verlag, 2008, pp. 516–541. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_22. URL: http://dx.doi.org/10.1007/978-3-540-70592-5_22.

[22] Intel. *Intel©Pentium D Processor Product brief.* URL: http://www.intel.com/content/www/us/en/processors/pentium/pentium-d-processor-brief.html (visited on 05/19/2015).

[23] *Jython: Python for the Java platform.* 2015. URL: http://www.jython.org/ (visited on 06/07/2015).

[24] *Kotlin homepage.* 2015. URL: http://kotlinlang.org/ (visited on 06/07/2015).

[25] J. M. Lucassen and D. K. Gifford. "Polymorphic Effect Systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 47–57. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73564. URL: http://doi.acm.org/10.1145/73560.73564.

[26] *Multi-core processor*. URL: http://en.wikipedia.org/wiki/Multi-core_processor (visited on 06/02/2015).

[27] Flemming Nielson and Hanne Riis Nielson. "Type and effect systems". In: *Correct System Design*. Springer, 1999, pp. 114–136.

[28] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002. ISBN: 0-262-16209-1.

[29] Marc Stiegler. *The E Language in a Walnut*. URL: http://www.skyhunter.com/marcs/ewalnut.html#SEC11 (visited on 06/06/2015).

[30] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. "Why Do Scala Developers Mix the Actor Model with Other Concurrency Models?" In: *Proceedings of the 27th European Conference on Object-Oriented Programming*. ECOOP'13. Montpellier, France: Springer-Verlag, 2013, pp. 302–326. ISBN: 978-3-642-39037-1. DOI: 10.1007/978-3-642-39038-8_13. URL: http://dx.doi.org/10.1007/978-3-642-39038-8_13.

[31] UPPAAL team. *UPPAAL About*. URL: http://www.it.uu.se/research/group/darts/uppaal/about.shtml (visited on 04/06/2015).

[32] Xtext Development Team. *Xtext - Language Development Made Easy!* URL: https://eclipse.org/Xtext/ (visited on 06/07/2015).

[33] *The Groovy programming language*. 2015. URL: http://www.groovy-lang.org/ (visited on 06/07/2015).

[34] *The Scala Programming Language*. 2015. URL: http://www.scala-lang.org/ (visited on 06/07/2015).

[35] *Thorn homepage*. 2015. URL: http://thorn-lang.org/ (visited on 06/07/2015).

[36] *TIOBE Software: Tiobe Index*. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html (visited on 05/19/2015).

[37] John A Trono. "A new exercise in concurrency". In: *ACM SIGCSE Bulletin* 26.3 (1994), pp. 8–10.

[38] Vasco T Vasconcelos, L Aceto, et al. "Sessions, from types to programming languages". In: *Bulletin of the EATCS* 103 (2011), pp. 53–73.

[39] Wikipedia. *Therac-25*. URL: http://en.wikipedia.org/wiki/Therac-25 (visited on 06/04/2015).

Appendix

Appendix A

Appendix

A.1 Grammars

```
1 grammar dk.homestead.joelang.spec.Spec with org.eclipse.xtext.common.Terminals
2
3 generate spec "http://www.homestead.dk/joelang/spec/Spec"
4
5 import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes
6
7 Specification:
8     'protocol' name=ID '{'
9         msgDefs += MessageDefinition+
10    '}'
11 ;
12
13 MessageDefinition:
14    isInitial ?= ('init')? name=ID '('
15        (
16            parameters += MessageParameter
17            (',' parameters += MessageParameter)*
18        )?
19    ')' (
20        '=>' leadsTo += ProtocolStep
21        (',' leadsTo += ProtocolStep)*
22    )?
23 ;
24
25 ProtocolStep:
26    asOutput?=('^')? target=[MessageDefinition]
27 ;
28
29 MessageParameter:
30    type=[jvmTypes::JvmFormalParameter]
31 ;
32
33 /* The normal ID terminal allows '^' as a first character. This caused some
34  * ambiguity. We introduce our own, identical to that of Xtext, with the one
35  * exception of the hat character.
36  */
37 terminal ID:
38    ('a'..'z'|'A'..'Z'|'$'|'_') ('a'..'z'|'A'..'Z'|'$'|'_'|'0'..'9')*;
```

```
1 grammar dk.homestead.joelang.actor.JoeLang with
  org.eclipse.xtext.common.Terminals
2
3 generate joeLang "http://www.homestead.dk/joelang/actor/JoeLang"
4
5 import "http://www.homestead.dk/joelang/spec/Spec" as Spec
6 import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes
7
8 Actor:
9     'actor' name=ID
10        (hasProtocol?='understands' protocols+=UnderstoodProtocol
11                                    (',' protocols+=UnderstoodProtocol)*)?
  '{'
12        (methods+=StepImplementation | fields+=Field)*
13    '}'
14;
15
16UnderstoodProtocol:
17    negatedProt?=('^')? protocol=[Spec::Specification]
18;
19
20StepImplementation:
21    name=ID '(' ((parameters+=Parameter) (',' parameters+=Parameter)*)? ')'
22        body=BlockExpression
23;
24
25BlockExpression:
26    {BlockExpression}
27    '{' (expressions+=Expression)* '}'
28;
29
30PrintCall:
31    'print' '(' toPrint=STRING ')'
32;
33
34// Should handle both fields and methods.
35/* Validate ID in regards to target during validation. */
36MessageCall:
37    (target=[MessageTarget]) '.' message=ID invocation=FeatureInvocation?
38;
39
40MessageTarget:
41    Field | LoopIterator | MessageSender
42;
43
44MessageSender:
45    name='sender'
46;
```

```
47
48 FeatureInvocation:
49     {FeatureInvocation}
50      '('
51         (
52             parameters+=ConcreteMessageParameter
53             (',' parameters+=ConcreteMessageParameter)*
54         )?
55     ')'
56 ;
57
58 ConcreteMessageParameter:
59     {ConcreteMessageParameter}
60     INT | STRING | MessageSender | ActorConstruction | FieldReference
61 ;
62
63 VarReference:
64     ref=[VarDeclaration]
65 ;
66
67 FieldReference:
68     ref=[Field]
69 ;
70
71 Field:
72     type=[Actor] (array?='[' ']')? name=ID ';'
73 ;
74
75 Parameter:
76     type=[Actor] name=ID
77 ;
78
79 /* The normal ID terminal allows '^' as a first character. This caused some
80  * ambiguity. We introduce our own, identical to that of Xtext, with the one
81  * exception of the hat character. *
82  */
83 terminal ID:
84     ('a'..'z'|'A'..'Z'|'$'|'_') ('a'..'z'|'A'..'Z'|'$'|'_'|'0'..'9')*;
85
86 // These expressions were mostly drawn from simple expressions package in
   antlr.
87 IfCondition:
88     'if' '(' condition=Expression ')' '{'
89     then+=Expression*
90     '}' ('else' '{'
91         else+=Expression*
92         '}'
93     )?
```

```
 94 ;
 95
 96 ForLoop:
 97     'for' '(' (iterator=LoopIterator ':' list=[Field]) ')' '{'
 98         body+=Expression*
 99     '}'
100 ;
101
102 ForNumLoop:
103     'fornum' '(' from=INT '..' to=INT ')' '{'
104         body+=Expression*
105     '}'
106 ;
107
108 LoopIterator:
109     name=ID
110 ;
111
112 Expression returns Expression:
113     AndExpression ( {OrExpression.left = current} '||' right = AndExpression
    )*
114 ;
115
116 AndExpression returns Expression:
117     Comparison ( {AndExpression.left = current} '&&' right = Comparison )*
118 ;
119
120 Comparison returns Expression:
121     PrefixExpression ( {Comparison.left = current} operator =
    ('=='|'<='|'>=') right = PrefixExpression)?
122 ;
123
124 PrefixExpression returns Expression:
125     {NotExpression} '!' expression = Atom
126   | Atom
127 ;
128
129 Atom returns Expression:
130     NumberLiteral
131   | MessageCall
132   | PrintCall
133   | IfCondition
134   | ForLoop
135   | ForNumLoop
136   | ActorConstruction
137   | Assignment
138   | VarDeclaration
139 ;
```

```
140
141 VarDeclaration:
142     type=ID name=ID
143 ;
144
145 ActorConstruction:
146     'new' actor=[Actor] invocation=FeatureInvocation
147 ;
148
149 Assignment:
150     feature=[Field] '=' (=>complex=[Parameter] | simple=SimpleAssignment)
151 ;
152
153 SimpleAssignment:
154     INT | STRING | ActorConstruction
155 ;
156
157 NumberLiteral:
158     value = INT
159 ;
```

A.2 Alternative Santa Claus Implementation

Listing 23 The Protocols used in a queued Santa Claus implemented in `Joe`.

```
1 protocol ElfQ {
2   ElfQ() -> Problem
3   Problem() -> ^Solution
4   Solution() -> ElfQ
5 }
```

```
1 protocol DoorProtocol {
2   DoorProtocol() -> ^Knock
3   Knock() -> Answer
4   Answer() -> DoorProtocol
5 }
```

```
1 protocol ReindeerQ {
2   ReindeerQ() -> Home
3   Home() -> ^Deliver
4   Deliver() -> ReindeerQ
5 }
```

Listing 24 The Actors of Elf and Reindeer. Like their one-to-one counterparts, they are quite simple.

```
1 actor QueueElf understands ^ElfQ {
2   QueueElf(ProblemQueue q) {
3     q.Problem()
4   }
5
6   Solution() {
7     sender.Problem()
8   }
9 }
```

```
1 actor QueueReindeer understands ^ReindeerQ {
2   QueueReindeer(HomeQueue q) {
3     q.Home()
4   }
5
6   Deliver() {
7     sender.Home()
8   }
9 }
```

Listing 25 The queued implementation of Santa. Note how this Actor does not note what kind of queue is messaging it, only responding along the established Protocol steps.

```
actor QueueSanta understands ^DoorProtocol {
  QueueSanta() {}

  Knock() {
    sender.Answer()
  }
}
```

Listing 26 The two queue implementations bear the brunt of the logic of this implementation. Individually, they mimic the one-to-one version of `Santa`'s lists of Actors to respond to.

```
actor HomeQueue understands ReindeerQ, DoorProtocol {
  QueueSanta santa

  QueueReindeer[] reindeer

  HomeQueue(QueueSanta s) {
    santa = s
  }

  Home() {
    reindeer.add(sender)
    if (reindeer.size() >= 3) {
      santa.Knock()
    }
  }

  Answer() {
    for (r : reindeer) {
      r.Deliver()
    }
  }
}
```

```
actor ProblemQueue understands ElfQ, DoorProtocol {
  QueueSanta santa
  QueueElf[] elves

  ProblemQueue(QueueSanta s) {
    santa = s
  }

  Problem() {
    elves.add(sender)
    if (elves.size() >= 3) {
      santa.Knock()
    }
  }

  Answer() {
    for (e : elves) {
      e.Solution()
    }
  }
}
```

Summary

This thesis is the conclusion of a project that has run over the course of two semesters. It carries on from the authors' prespecialisation, "Sorting and Synchronising", in 2014. Where the previous report delved into the difficulties of designing safe concurrent and parallel implementations across a number of programming languages, this thesis goes one step further, defining a new programming language, designed to address these issues while retaining a syntax that should be well-known to many programmers.

`Joe` is an inherently concurrent, actor-based programming language, with a syntax very similar to `Java`. The similarity is intentional with a few notable exceptions taking the place of `Java`'s classes and interfaces. *Actors*, a parallel to classes, represent a concurrent agent, capable of receiving and sending messages to other Actors. The concept of *Protocols* takes the place of interfaces. A Protocol defines a number of message types as well as the series of messages (or steps) that constitute a correct usage of the Protocol.

In order to ensure the correctness of communication between Actors, they are translated to a format compatible with the UPPAAL verification tool. This is accomplished by constructing a simple Control Flow Graph from the output of the Xtext framework, which is used in the development of `Joe`.

At the time of writing, a subset of `Joe` was implemented, in order to implement a solution to Trono's Santa Claus problem, allowing us to draw comparisons with similar implementations in the prespecialisation. We find that the `Joe` implementations are shorter and more concise compared to those of `Java`, `X10`, `Clojure` and `Erlang`.

Although we have managed fundamental strides towards verification of Actor communications, key elements are missing from the currently generated control flow graphs, hampering any broad conclusions.

Despite of its shortcomings, we consider `Joe` a promising example of what a modern programming language with deeply ingrained concurrency should be.