Case for Support

FORSE - Formally-based tool support for Erlang development

Principal Investigators:
Professor John Derrick, University of Sheffield
Professor Simon Thompson, University of Kent

Academic Collaborator:
Lars-Ake Fredlund, Swedish Institute of Computer Science (SICS)

Industrial Partners:
T-mobile UK Ltd
Erlang Training and Consulting Ltd

Keywords: Erlang, functional programming, refactoring, process algebras, testing.

Part 1 - Previous research

Staff at the Universities of Sheffield and Kent are well-placed to carry out this work, and we have a track record in the key areas in this proposal. These themes include functional programming, verification and formal methods, and testing.

Functional programming. Work at the University of Kent goes back over 20 years beginning with the development of Miranda, through work on type theory, Haskell, and techniques and tools for functional programming.

Techniques and tools. Functional programming is a different paradigm from OO or procedural programming, but it shares a number of aspects with other paradigms. Principally, the control of complexity is an issue irrespective of the language involved, and work by staff and students at Kent has investigated *software engineering* aspects of functional programming. Projects have included design [35], software measurement and visualisation [33], and refactoring functional programs [32].

Other work in the department has explored the use of the functional paradigm in 'document centred programming' in the VITAL project [29] and graphical extensions to functional programming languages.

Refactoring. Kent is the home of the EPSRC-sponsored *Refactoring Functional Programs* project(GR/R75052, 2002–5) [32, 46], website: `http://www.cs.kent.ac.uk/projects/refactor-fp/`. The principal deliverable of the refactoring project is the **Ha**skell **Re**factorer, HaRe. This tool, which covers the complete Haskell 98 language, respects program layout in refactoring code and integrates with the most used Haskell IDEs, it is now in its third release.

Haskell. Simon Thompson is the author of one of the standard texts on functional programming in Haskell, currently in its second edition [45]. He has also worked on reasoning about functional programs in lazy functional programming languages such as Miranda [43] and Haskell [42] as well examining the verification of programs using particular features of these languages, such as input/output [39] and "laws" [40]. He has applied functional programming ideas to semantic description in [44].

Formal methods and verification. Work on formal methods has centred on specification languages, both process algebraic and state-based, their relationship and verification and development. Of particular relevance is the following.

Process algebras. Work on process algebra has included applications in distributed systems, model checking and links to other formalisms. Applied work has included the use of process algebras in the ODP standard [26, 10], subtyping [11] and specification and analysis [21, 12]. Theoretical work has included integrations of process algebras with other notations [36, 37], the semantics of refinement in process algebras [36, 37, 25].

Model checking for Erlang. As detailed in the case for support, we have been involved in collaborative work concerning verification support for Erlang. The approach has been one where Erlang/OTP is translated into the μCRL process algebra, upon which model-checking is performed. This work has been successfully applied to a number of application areas (see, e.g., in conjunction with Ericsson and IT-university in Gothenburg, Sweden [4], Universidad de A Coruna, Spain and the cable and telecommunications company R, Galicia, Spain [7]) in order to test its applicability. (More detail is included in Part 2.)

Other work on model checking has included model checking for stochastic automata [18, 15], deadlocks in timed process algebras [14]. Both departments have wide ranging experience in the use of a variety of model checking systems such as Caesar/Aldebaran, Uppaal, FDR etc.

Testing. Work by the proposers has included that on deriving tests from state-based specifications, e.g., we have shown how refinement can be used to simplify the testing and test case generation process [22, 24]. Both Kent and Sheffield are part of the EPSRC FORTEST (FORmal methods and TESTing) network which is just nearing completion, and Sheffield is home to extensive work on specification based testing.

The proposers have an excellent track record of successful management of EPSRC and other projects. Feedback from their last EPSRC final report (GR/L28890/01) noted:

> *Exactly what one would expect from the given team: professional and thorough, with excellent dissemination. This team has an excellent reputation for cost effectiveness.* (Referee 6463PV)
>
> *Significant fundamental innovation. Excellent achievements in training.* (Referee 77NN8H)

John Derrick is Professor of Formal Methods at Kent and will take up a Chair at Sheffield from January 2005, he will subsequently hold a visiting position at Kent. His current interests are in applied formal methods, particularly distributed systems, and include work on refinement, ODP, applications of UML, policy specification, performance and QoS. He has published extensively within this area, including over 100 journal and conference papers and two research monographs [23, 13]. He is a member of several programme committees (Z Users meeting, FMOODS, IFM) and has edited a number of special journal issues. He has extensive experience of applying formal modelling and analysis techniques, particularly within the DS domain, and funded projects include the following (in addition to those mentioned above): PROST (DTI) *Study on Conformance Testing for ODP*; (DTI) *A study into Formal Description Techniques for Object Management*; (EPSRC) *Cross Viewpoint Consistency in Open Distributed Processing*; FORMOSA (EPSRC/DTI) *Formalisation of the ODP Systems Architecture*; (British Telecom) *Type Management in Distributed Systems*; (EPSRC) *ODP Viewpoints in a Development Framework*; (EPSRC) *A Specification Architecture for the Validation of Real-time and Stochastic Quality of Service*; (EPSRC) *Design Support Environments for Distributed Systems*; (EPSRC) *FORCES*; (EPSRC) *A Constructive Framework for Partial Specification*, (British Telecom) *Scheduling policies for active networks*, and (EPSRC) *RefineNet*.

Simon Thompson is Professor of Logic and Computation and Director of the Computing Laboratory at the University of Kent. He has considerable experience of the theory and practice of functional programming. He has received funding from EPSRC (and its predecessors) for research in functional programming and computational logic; recent grants include *Integrating Computer Algebra and Reasoning: Incorporating a logic in the Axiom system* (GR/M37851/01, 1999–2002) [34] and *HaRe* discussed earlier. His work on refactoring and other aspects of functional programming is discussed earlier.

His work in logic includes a long-standing interest in proof and program verification in constructive type theory and he is the author of one of the graduate texts in this field [41]. His work on diagrammatic reason with the University of Brighton is supported by EPSRC under *Reasoning with Diagrams* (GR/R63509, 2002–2005) [38]. He also has interests in the application and theory of temporal logics [17], and in logics for multimedia [16].

Huiqing Li is exceptionally well-placed to work on the refactoring aspects of this grant. She graduated with a Master's degree in computing science and engineering in 1995, and worked at software engineering and programming language processing since then. She is the research Assistant on the project *Refactoring Functional Programs* (GR/R75052, 2002–5), and as such she is the principal architect and implementor of the Haskell Refactorer, HaRe, [32]. She is also working towards a PhD which will report on both the implementation of HaRe and the formal verification of various aspects of refactoring for functional programs.

Clara Benac Earle is the ideal candidate to work on the verification aspects of this grant. She graduated with an MSc from the University of Madrid, and worked at Ericsson on prototype verification tool support for Erlang. With Thomas Arts she worked on model checking Erlang code, developing the tool *etomcrl* (see also http://sourceforge.net/projects/etomcrl), and applying it to the AXD301 code [3]. Subsequently she undertook a PhD at the University of Kent, which will be completed in September 2004. This has further developed the approach, and some of the preliminary findings are published in [4, 5] in addition to [3].

Part 2 - The Proposal

1 Executive Summary

The problem. Customers need to be able to reply on IT-based services delivered at an economic rate. Moreover, the requirements on these systems continually grow. Rather than being monolithic programs running on mainframe computers, applications are now expected to be concurrent, distributed, fault-tolerant whilst subject to real-time constraints.

For business critical applications, reliability and robustness are crucial, and current methods for achieving this are expensive and limited. The need for appropriate processes and tools to help engineer complex systems is clear. The research proposed here will address this current gap.

Erlang. The programming language Erlang is designed with precisely this sort of modern system in mind. Erlang has a unique combination of simplicity of program expression – because of the language's high-level features – and expressivity and efficiency of program code. Erlang is a concurrent functional language with specific support for the development of concurrent, distributed systems with soft real-time requirements. Central to modern Erlang system development is the Open Telecom Platform (OTP): an architecture to construct complete fault-tolerant systems. OTP is realised in a library of components which embody a number of powerful, high-level, design patterns.

The success of Erlang. Erlang has been a significant success, having been used in a variety of business critical applications. Erlang was originally conceived within Ericsson, but now has a wider enthusiastic developer community, and is used in companies of all sizes, most commonly small teams within SMEs. Its domain of application including telecoms and computer telephony but also covers banking, TCP/IP programming, 3D-modelling and more. With the trend to ubiquitous systems Erlang will grow more and more important.

However, while Erlang's characteristics have been used to shorten development time, ensuring the reliability of systems requires extensive testing and verification, at the expense of delivery time. The problem thus is one of finding a more reliable – yet at the same time more agile – engineering process.

Our proposal: next generation tools and techniques. It is acknowledged in the Erlang developer community that there is a clear need for next generation tools and technologies:

model checking: help to establish explicit properties of design patterns that guarantee correctness of the system;
refactoring: make it possible to change the code and be able to check whether pre-defined properties still hold;
testing: make this more effective, reducing testing time whilst increasing test coverage.

This project will deliver precisely those tools and techniques.

Ground-breaking research – industrial tensioning. Delivering these tools presents us with some serious research issues: the mix of concurrent, distributed and real-time aspects in Erlang is unique. Moreover, these elements are combined with the architectural principles of the OTP library, specifically designed to support fault-tolerance through its design patterns. At the same time, the language has sound semantic underpinnings which offer the opportunity for the well-founded and effective engineering of tool support.

Our aim is to help facilitate a step change in practice by making such support available. To do so we will link our work into industrial practitioners by working with industrial Erlang users in order to develop, deploy and test our ideas. Researchers will work with our partners to design tooling on the basis of proven industrial needs; prototype tools will be deployed in engineering solutions to industrially-based case studies.

In doing this work we will build on our internationally recognised research in model checking, verification, refactoring and testing. Moreover, we have identified named researchers who bring to the project exactly the right skills and experience to allow the team to 'hit the ground running'.

2 Background

Erlang

Erlang [1] is a functional programming language with support for concurrency, scheduling, distribution and memory management. It supports both concurrency and distribution using light-weight processes and asynchronous message passing. It has been used to implement some substantial business critical applications, e.g., the Ericsson AXD 301 high capacity ATM switch [8] which at over a million lines of Erlang code is used to implement the backbone network in the UK.

Erlang is available under an Open Source licence from Ericsson, and its use has spread to a variety of sectors. Applications include TCP/IP programming (HTTP, SSL, Email, Instant messaging, etc), web-servers, databases, advanced call control services, banking, 3D-modelling. This user base is now Europe-wide with firms in the UK (both big and small), France, Sweden, Germany amongst others.

Some aspects of Erlang make it comparable to Java. Both languages aim to work on a variety of devices (since they both compile into virtual machine code) and platforms. They each aim to be general purpose, and have support for distribution and concurrency. It is in their concurrent features that they differ most markedly. Erlang provides support for massively concurrent systems through its own process management system, and it is not untypical for an Erlang system to support tens of thousands of concurrent processes. Java, on the other hand, supports threading. Java threading is notoriously tricky; indeed, an early version of the Sun tutorial on threads suggested that 'The first rule of threads is this: avoid them if you can'. Erlang and OTP, by contrast, encourage highly concurrent architectures and moreover support them in efficient implementations. Of equal importance is the fact that Erlang's processes have a rigorous semantics.

Design patterns

Erlang/OTP software is usually written according to strict design patterns that make extensive use of software components. Encapsulated in the extensive OTP library are a variety of design patterns, each of which is intended to solve a particular class of problem. Solutions to each such problem come in two parts. The generic part is provided by OTP as a library module and the specific part is implemented by the programmer in Erlang. Typically these specific callback functions embody algorithmic features of the system, whilst the generic components provide for fault tolerance, fault isolation and so forth.

Fault tolerance is provided via a supervisor component in which the runtime system provides a communication mechanism in the presence of software or hardware failure. A supervisor process monitors this communication and uses one of a number of policies by which to recover from a crashed process.

In addition to generic servers and supervisors, other generic components include finite state machines, event handlers and applications and these considerably simplify the building of systems. Erlang's simplicity and these features make it attractive to programmers, it also makes it particularly suitable for formal support through refactoring, model checking and testing.

Refactoring

Refactorings are source-to-source program transformations that change program structure and organisation, but not program functionality. Documented in catalogues and supported by tools, refactoring provides the means to adapt and improve the design of existing code, and has thus helped to address long-standing problems in software maintenance while also enabling the trend towards modern agile software development processes. Two typical refactoring scenarios are: adapting the structure of an existing code base to prepare for adding functionality or fixing a bug; cleaning up a code base after extensive functional changes.

Current work on refactoring has concentrated on refactoring sequential aspects of (typically) OO programs. The complicated semantics of Java, Smalltalk and other languages mean that the correctness of many refactorings can only be ensured through extensive testing of the refactored code. The Kent-based, EPSRC-supported project on *Refactoring Functional Programs* has built a tool implementing many refactorings for Haskell programs [32, 46]. Because of the clean semantics of functional languages it is possible to guarantee the correctness of each refactoring – under well-defined conditions – without recourse to testing. The solid semantic foundations of Erlang will allow us a 'quick win' in transferring this work to Erlang from Haskell.

It is not simply a matter of transferring results from Haskell, however. Refactoring for Erlang presents two novel research challenges. First, concurrency is central to the Erlang ethos, yet refactoring in the presence of concurrency is thus far unexplored. We expect to deploy our refactoring experience in this direction. Secondly, the OTP library presents a particular architectural framework and this raises a series of questions: How should OTP programs be refactored? What are the particular problems associated with refactoring Erlang programs which appear – as a callback or a state-transformer, for example – within an OTP context?

Model Checking

Model checking is an automatic formal verification technique where a property is checked over a finite state (concurrent) system, and it has been very successfully applied on hardware descriptions and on specifications (e.g., automata, process algebras). Model checking applied at the program level (i.e., as *program* verification as opposed to *specification* verification) brings extra non-trivial challenges, and abstractions are often necessary in order for programs to be amenable to analysis.

In this proposal we seek to build upon our existing work on model checking Erlang. In collaboration with partners, we have worked on tool support for verification which includes a number of components: translation, state-space generation and model checking. In order to verify properties, Erlang code is translated into a process algebraic specification given in μCRL, to which standard model checking tools are applied (e.g., Caesar/Aldebaran) [48]. The translation from Erlang to μCRL is performed in two stages. First, a source to source transformation is applied, resulting in Erlang code that is optimised for the verification, but has identical behaviour (i.e., we refactor). Second, this output is translated to μCRL. The tool currently produces translations of the callback modules of the generic servers and supervisors (these two normally accounting for over 80% of OTP code).

The translation is quite involved due to particular language features in Erlang. For example, Erlang makes use of higher-order functions, whereas μCRL is 1st order; Erlang is dynamically typed, but μCRL is statically typed; in Erlang communication can take place in a computation, in μCRL it cannot. However, μCRL is sufficiently close that such a translation is feasible, and model checking on it computationally traceable even if the translation is involved. The prototype tool, *etomcrl*, is described in [5], and case studies using it in [4, 3].

Other related work in this area includes PathFinder [30] and Bandera [20] which consider the problem of verifying Java code. Our work is based on a similar idea, except that we use the Erlang design patterns in order to obtain smaller state spaces. In terms of Erlang relevant work includes that by Huch [31], however, here the level of abstraction chosen meant that key properties concerning data could not be verified, and our work has sought to rectify this [2].

Testing

There is a large amount of theory and practice on derivation of test suites from specifications, e.g., testing from process algebraic specifications and testing asynchronous languages such as SDL.

Within Erlang the OTP test environment provides facilities for designers to write their own test cases which can be automatically executed overnight. The OTP test suite currently comprises over 3,000 such test cases. However, OTP only provides for particular approaches to testing (e.g., traffic load tests), and is targeted towards an application slant (telecoms).

More general approaches to test generation include Erlang/QuickCheck, which is a library for random testing of Erlang programs against specifications [6, 19]. Similar ideas have also been considered in [9], where a technique for generating test suites from state machine specifications (written in a syntax close to Erlang) is derived. However, none of these approaches provides a general purpose way of systematic test generation, this is what we wish to do and will exploit some of the work on abstraction and refactoring to do so.

3 Workpackages

In this proposal we aim to build an environment containing a range of tools designed to support the refactoring, verification and testing of Erlang programs.

Our *vision* for refactoring in Erlang is to make machine-supported refactoring second nature to the Erlang programmer. Refactorings can be performed in a speculative way at minimal cost, and undone just as easily. Transition to OTP from 'raw' Erlang will also be given as much automated support as is practicable.

Our *vision* for the verification technology for Erlang is based upon model checking key properties against a process algebraic abstraction of the Erlang code. We know its feasibility, we now need to tackle some fundamental research issues. These include extending the translation to cover the fault-tolerant and real-time aspects of OTP; to scale up the translation to other design patterns, and to use refactoring to optimise the translation.

Our *approach* to verification and refactoring will be brought together in the final workpackage when we apply them to the issue of test generation.

Two prerequisite for these components are the basic infrastructure, which is provided in the first workpackage, and a semantic foundation for our work, and this forms the second workpackage.

The project requires research innovation in each of its strands. At the same time we will work closely with industrial partners, who will influence the project throughout its lifetime by providing case studies, design input, dissemination and evaluation of the work in its intended context.

We believe the strength of the work lies in it not just being a single neat idea, but work in which several components have already been tested and found useful, and such that their development and integration offers hope for significant impact. The adventure and risk (and excitment!) lie in the realisation of the sum of the parts together.

WP1 - Project Infrastructure

The project will deliver an integrated set of tools to manipulate and process Erlang programs. Underlying the toolset is a common Erlang infrastructure, comprising 'front end' components including a parser that preserves source information such as code layout and comments; a pretty-printer that reproduces source code from abstract syntax trees; type and module analysers; a library of traversal operations on syntax trees to support program analysis and transformation; and tooling for testing.

Workpackage 1 will deliver this common infrastructure. Much of this exists already in a fragmentary form: the workpackage will involve selecting between the various candidates, integrating the best available and developing those components which are not currently available. This may involve porting into Erlang components already developed for other languages.

Development of this package will allow the project research assistants to get up to speed with Erlang programming. The RAs will also gain experience in testing and refactoring Erlang programs, which will be valuable in WPs 3, 4, 5 and 11. Integration work will be guided by the industrial collaborators, and the platform developed will be released in Open Source format, to promote its adoption and further enhancement by the Erlang developer community.

Development and maintenance of this workpackage will continue throughout the lifetime of the project.

WP2 - Semantic Foundations

Fredlund has defined an operation semantics for Erlang, which will give the formal underpinning for the work of the project. The semantics can be used to define conditions under which two programs are equivalent; to form the basis for the formal generation of test data and for the translation and abstraction of certain aspects of Erlang into μCRL.

The OTP is an Erlang library, and thus described by Fredlund's semantics. However, the principles of OTP design impose a strong set of constraints on systems which use it. For example, fault tolerance is enforced by structuring processes into a supervision tree in which parent nodes (and only parent nodes) are responsible for starting and restarting child processes. In giving the semantics of processes in this context it is therefore not necessary to monitor failure of all other processes in the system (which is possible in the case of 'raw' Erlang). We will develop, in collaboration with Fredlund as a Visiting Researcher, a simplified operational semantics of OTP, based on his original Erlang semantics.

The existing translation of a subset of Erlang/OTP into μCRL is not sufficient as a general semantic underpinning since it abstracts away aspects of Erlang program behaviour in order to obtain tractable behaviour of model-checking. Therefore, in this workpackage we will verify that the OTP operational semantics is consistent with the process-algebraic translation.

WP3 - Basic Refactorings

We have successfully developed the HaRe refactoring tool for Haskell, which is a sequential functional programming language. The refactorings in Hare were written in two stages: first we refactored single-module Haskell programs, and then we extended these to refactorings to multiple-module projects. Hare is itself written in Haskell.

We will follow a similar approach in building the first phase of the Erlang tool, transferring the insight and technology from the HaRe project to Erlang. We observe that sequential Erlang refactorings are not without interest, since OTP programs can 'wire together' sequential programs in particular stylised forms, and so refactoring the application-specific parts of a system will have a strong sequential component.

We expect that our experience with Haskell refactorings will allow us to complete this work package quickly, and so spend most of the project on Erlang-specific aspects of refactoring.

WP4 - Refactoring Concurrency

Concurrency is essential to Erlang system development. Whilst it is acknowledged that refactoring concurrent code in, say, Java, presents a substantial challenge [27], the message-passing mechanism of Erlang promises to be more tractable. Our initial target in this work package is to design and implement refactorings for core Erlang concurrency.

Building on top of these we will then investigate common patterns of concurrency and fault-tolerance in substantial Erlang projects. We will devise means for building up "refactorings-in-the-large" from the basic stock of concurrent refactorings developed earlier.

WP5 - Refactoring within OTP

The separation of Erlang into a base language and a set of design patterns (behaviours) encoded in the standard OTP libraries shapes the practice of Erlang programmers. The majority of Erlang code will be particular, application-specific, functionality which is intended to be passed to one of the generic operations within OTP. The architecture of OTP requires that the application-specific code behaves in particular ways (and not in others).

This impacts on refactoring. Some of the refactorings developed in earlier work packages will only be valid in an OTP context if programs meet additional conditions; other refactorings will become possible because of the constraints required by OTP. In this work package we will articulate and implement these additional conditions and refactorings, so that the tool can successfully support the OTP developer.

WP6 - Fault Tolerance

Erlang has fault tolerance mechanisms based around a supervision tree which restarts crashed processes according to one of a number of policies (one-for-one, one-for-all etc). The existing translation of Erlang to μCRL is sufficiently robust to deal with a large enough part of the language to make it applicable to serious examples. However, it is currently limited by not being able to deal with the fault tolerant aspects of the language.

In this workpackage we will extend the translation into μCRL to produce a translation which models the effect of the supervision tree and various policies, in an untimed context. The translation algorithm will be implemented as part of the toolset, and be evaluated by the application of model checking to these translations, checking a range of properties such as process liveness under faulty behaviour.

WP7 - Real-time aspects

Erlang is equipped with real-time capabilities, such as timeouts, which are used in the generic server and finite state machine OTP design patterns, for instance. Timing is also used in the fault tolerant policies, which allow a maximum number of restarts in particular time intervals, given as parameters. Modelling this behaviour is important since if the maximum is exceeded, the crash propagates up the supervisor tree and ultimately to the top application.

The current Erlang to μCRL translation is untimed, and here we will look at incorporating timing information, in order to model both timeouts and the full fault tolerant behaviour.

To do this we will use recent work on timed extensions to μCRL [28, 47]. An extensive theory exists for these extensions (completeness, bisimularity, complete axiomatisations etc), and as noted in [47] it is possible to use verification tools that exist for μCRL for the analysis of timed processes.

We will take this as our starting point and first extend the translation to timeouts, with an embedding on which untimed μCRL tools can be used. We will then extend the translation of the fault tolerant policies to include their restart strategy in terms of crashes per time interval and propagation up the supervision tree.

The latter opens up the possibility to explore settings of the restart parameters in order to optimize locality of the crashes vs their propagation up the supervision tree. Currently default values are provided in Erlang and there is little attempt to model or simulate the consequences of the other particular choice of parameter. Our analysis and simulation of the process algebraic abstraction will tackle this issue.

WP8 - Scaling up the translation

The extension to fault tolerant and real-time behaviour will significantly enhance the applicability of the approach, but further work is necessary in order to scale-up the application of verification technology. In particular, we need to extend the translation to other OTP design patterns and tackle the issue of dynamic process creation in the translation of the supervisor.

Task 8.1 - other OTP design patterns. In addition to generic servers and supervisors, OTP defines applications, finite state machines and event handlers. Hooks exist for these in the current tool, and this task is to extend the translation to these design patterns (using input from our industrial partners to assess relative importance). This is relatively straightforward, but will enhance the coverage of the approach.

Task 8.2 - dynamic processes. In a supervisor Erlang processes can, in general, be generated dynamically. However, to avoid a potentially infinite state-space we avoid dynamic process creation in μCRL by generating μCRL specifications for fixed configurations. This needs generalising, and we seek to see how this can be supported in the verification framework. To do this we will (1) look at circumstances where dynamic process creation still gives a finite state-space, (2) consider whether additional verification can generalise results from fixed to arbitrary configurations. For the latter, we will consider how we can apply both theorem proving as well as techniques such as symmetry, data-independence and compositionality results.

WP9 - Optimising the translation by refactoring

In this task we look at how we can use refactoring to optimize the source-to-source Erlang transformation currently used in the model checking tool.

A number of Erlang features need manual treatment at present. For example, the translation step needs various higher-order functions to be re-written in order that they can be expressed in the process algebra. Similarly, two aspects of pattern matching are not supported. First, one needs to eliminate priority re-writing (priorities not being supported in μCRL). Second, code where no match is found is not currently supported, since such an Erlang process crashes, whereas this is not reflected in the μCRL semantics.

The tasks here then are to (1) use refactoring techniques to support the rewriting of higher-order functions, and (2) provide similar support for priority rewriting, and investigate approaches to translating code where no matches are found.

In addition, the size of the state space produced in the model-checking is dependent on the code structure. Thus we will also look at using refactoring to optimise the size of the LTS produced from the translation algorithm.

WP10 - Evaluation, iteration and deployment

Our aim is to link into industrial practice, in order to find out how best to deploy these technologies by working with real developers on real projects. Our industrial partners will help with deployment once the (1st generation) tools are built, and we will iterate on the basis of this experience.

There are a number of stages to this. To begin with the research staff will work with our partners to gain an understanding of the Erlang development process and an insight to the processes and tools currently used. This experience will be used in WP1 on tool integration, and also in WP3 as this needs an understanding of the most useful refactorings.

The project RAs will visit the industrial partners for a period of roughly three weeks in the first half year of the project, and will undertake more substantial visits, of, say, two to three months, in the second and third years of the project.

Our industrial partners will also help us to evaluate the work. Six monthly review meetings (open to all interested in Erlang development, not just the industrial partners) will be used to disseminate the work informally.

We will also use our partners, and other European Erlang developers, to look at wider questions. For example, we would like to investigate alternative process algebraic translations during the project. The issue here is that the translation affects in a fundamental way the properties that can be verified. This is shown by the choice of translation made by Huch [31] where, since data was abstracted to a non-deterministic choice, key properties couldn't be verified. We will thus also take a step back from the existing work to see whether different approaches to translation might offer gains for different problems that one wishes to verify.

WP11 - Testing

This workpackage constitutes the programmme of work for the PhD student supported by the project studentship.

Providing further automation of the testing process is necessary in order to support the work of both small developer teams as well as those developing large code bases.

Whilst there is some support for the testing process via OTPs test environment and the QuickCheck Erlang tool, there is considerable scope to provide further automation and support. We seek here to do three things:

- integrate the QuickCheck Erlang tool into the development environment.
- apply existing work on specification based testing to help automate the derivation of the individual test cases and their sequencing.
- apply refactoring techniques to enable tests and their specifications to be refactored.

The QuickCheck Erlang tool enables random tests to be generated from descriptions given as logical formulas over

Erlang terms. Our first task is to integrate this into the basic infrastructure.

The general problem of deriving non-random tests is harder, and there are significant challenges here due to the mix in Erlang of data, concurrency, asynchronous message passing and higher order function definition. However, relevant work on testing exists in a number of communities, and here we seek to use one approach. In particular, we will use the process algebraic translation used in model checking as the basis for test case generation to tackle issues of concurrency and temporal ordering.

To do this, we will use the inverse of the Erlang to μCRL translation to define test cases and test sequencing of the Erlang code. Defining an appropriate inverse will be the first task in achieving this, and will allow the use of testing facilities in the Caesar/Aldebaran toolset. The effectiveness of this will be assessed by generating tests for properties expressed as specifications in μCRL. The possibility then exists for integrating this approach with the random tests generated by the Erlang version of QuickCheck.

Finally, we aim to apply our refactoring techniques in order that tests can be refactored alongside the code base. For tests generated by QuickCheck we will also derive a means to refactor the test specification (given as a linear temporal logic property), so it is clear what tests have been executed. For tests generated via a process algebraic specification, we will use the equivalence defined in WP2 to derive a new process algebraic specification in addition to the refactored tests. The OTP test environment will be used as a case study for parts of this workpackage.

4 Relevance to Beneficiaries

Beneficiaries will include the functional programming and Erlang research and developer communities as well as those interested in verification and applications of model checking.

In particular we offer tangible benefits to the Erlang developer community, since there are clear potential economic gains to be made from exploiting formal methods technology to reduce the costs (in time and money) of ensuring reliability, and by offering additional routes to assurance where testing alone would be insufficient. Further details are given in section L of the accompanying EPSRC form.

5 Dissemination and Exploitation

Dissemination, collaboration and exploitation are important aspects of this proposal. Project deliverables will include

- tools, and tool plug-ins (disseminated via a project website and through the Erlang pages)
- demonstrator projects
- engagement with industrial practice
- verified and (formally) specified components (behaviours)
- scientific publications in conferences and journals

The latter include Journal of Functional Programming, Erlang User Conference, Formal Aspects of Computing, Formal Methods in Systems Design, FME, FORTE/PSTV, etc etc.

Existing contact will be maintained with the Erlang community. This includes continuing our existing collaborations with the IT-university in Gothenburg, Sweden and the Universidad de A Coruna, Spain, and these and other Erlang sites will be involved in the 6-monthly project review meetings. Visits to these sites are costed in the proposal.

6 Justification of Resources

This project builds upon considerable expertise available at Sheffield and Kent in the areas of functional programming and formal methods. We are lucky to have two candidates for the RA posts who bring a wealth of relevant experience with them. The project is also a true collaboration with links between the strands in a way that should be mutually beneficial to the overall work. We envisage close working to achieve the best results. **Equipment:** The equipment requested will support the Research Staff and PhD student. **Travel:** Support for travel is requested to support the dissemination and exploitation activities outlined above.

Bibliography

[1] J.L. Armstrong, S.R. Virding, M.C. Williams, and C. Wikström. *Concurrent Programming in Erlang.* Prentice Hall International, 1996.

[2] T. Arts and C.B̃enac Earle. Development of a verified distributed resource locker. In *International workshop on Formal Methods in Industrial Critical Systems*, July 2001.

[3] T. Arts, C. Benac Earle, and J. Derrick. Verifying Erlang code: a resource locker case-study. In L. Eriksson and P.A. Lindsay, editors, *Formal Methods Europe: Getting IT Right*, volume 2391 of *LNCS*, pages 184–203. Springer-Verlag, July 2002.

[4] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[5] T. Arts, C. Benac Earle, and J. J. S. Penas. Translating Erlang to mCRL. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004).* IEEE Computer Society Press, June 2004. To appear.

[6] T. Arts and J. Hughes. Erlang/QuickCheck. In *Ninth International Erlang/OTP User Conference*, 2003.

[7] T. Arts and J. J. S. Penas. Global scheduler properties derived from local restrictions. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 49–57. ACM Press, 2002.

[8] S. Blau and J. Rooth. AXD 301 – A new Generation ATM Switching System. *Ericsson Review*, 1, 1998.

[9] J. Blom and B. Jonsson. Automated test generation for industrial Erlang applications. In *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14. ACM Press, 2003.

[10] H. Bowman, E. A. Boiten, J. Derrick, and M. W. A. Steen. Strategies for consistency checking based on unification. *Science of Computer Programming*, 33:261–298, April 1999.

[11] H. Bowman, C. Briscoe-Smith, J. Derrick, and B. Strulo. On Behavioural Subtyping in LOTOS. In *FMOODS'97, Second IFIP International Conference on Formal Methods for Open*

Object-based Distributed Systems. Chapman and Hall, July 1997.

[12] H. Bowman, J.W. Bryans, and J. Derrick. Analysis of a multimedia stream using stochastic process algebra. *The Computer Journal*, 44(4):230–245, April 2001.

[13] H. Bowman and J. Derrick, editors. *Formal Methods for Distributed Processing, A Survey of Object-oriented Approaches.* Cambridge University Press, Cambridge, UK, September 2001.

[14] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronisation protocol using UPPAAL. *Formal Aspects of Computing*, 10(5-6):550–575, August 1998.

[15] H. Bowman, J.W.Bryans, and J. Derrick. Towards stochastic model checking with generalised distributions. In *UKPEW 2000, 16th United Kingdom Performance Engineering Workshop*, November 2000.

[16] Howard Bowman, Helen Cameron, Peter King, and Simon Thompson. Mexitl: Multimedia in executable interval temporal logic. *Formal Methods in System Design*, 22, 2003.

[17] Howard Bowman and Simon Thompson. A tableau method for interval temporal logic with projection. In *TABLEAUX'98, International Conference on Analytic Tableaux and Related Methods*, volume 1397 of *LNAI*. Springer-Verlag, 1998.

[18] J. Bryans, H. Bowman, and J. Derrick. Model Checking Stochastic Automata. *ACM Transactions on Computational Logic*, 4(4):452–492, October 2003.

[19] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM Press, 2000.

[20] J. C. Corbett, M. B. Dwyer, and J. Hatcliff. Bandera: a source-level interface for model checking Java programs. In *Proceedings of the 22nd international conference on Software engineering*, pages 762–765. ACM Press, 2000.

[21] J. Derrick. Timed CSP and Object-Z. In *ZB 2003: Formal Specification and Development in Z and B*, volume 2651 of *Lecture Notes in Computer Science*, pages 300–318. Springer, June 2003.

[22] J. Derrick and E. A. Boiten. Testing refinements by refining tests. In *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *LNCS*, pages 265–283. Springer-Verlag, 1998.

[23] J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z.* Springer-Verlag, 2001.

[24] J. Derrick and E.A. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, (9):27–50, July 1999.

[25] J. Derrick and E.A. Boiten. Combining component specifications in Object-Z and CSP. *Formal Aspects of Computing*, 13:111–127, May 2002.

[26] John Derrick, Eerke Boiten, Howard Bowman, and Maarten Steen. Viewpoints and consistency: translating LOTOS to Object-Z. *Computer Standards and Interfaces*, 21:251–272, August 1999.

[27] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Observations on the Assured Evolution of Concurrent Java programs. In *Proc. Workshop in Concurrency and Synchronisation in Java Programming.* ACM, 2004.

[28] J.F. Groote. The syntax and semantics of timed mCRL. Technical Report SEN-R9709, CWI, Amsterdam, 1997.

[29] Keith Hanna. Interactive Visual Functional Programming. In Simon Peyton Jones, editor, *Proc. Intnl Conf. on Functional Programming.* ACM, 2002.

[30] K. Havelund and T. Pressburger. Model checking Java Programs with Java PathFinder. *Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[31] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261–272. ACM Press, 1999.

[32] Huiqing Li, Claus Reinke, and Simon Thompson. Tool Support for Refactoring Functional Programs. In *ACM Sigplan Haskell Workshop*, 2003.

[33] The Medina Metrics Library. Available from `http://www.cs.kent.ac.uk/people/rpg/cr24/medina/`.

[34] Erik Poll and Simon Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems, 2000.* LNCS 1794, Springer-Verlag, 2000.

[35] Daniel J. Russell. *FAD: Functional Analysis and Design Methodology.* PhD thesis, University of Kent, 2000.

[36] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In *ICFEM'97*, pages 293–302. IEEE Computer Society Press, 1997.

[37] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.

[38] Gem Stapleton, John Howse, John Taylor, and Simon Thompson. What Can Spider Diagrams Say? In Alan Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Diagrammatic Representation and Inference*, 2004.

[39] Simon Thompson. Interactive functional programs: a method and a formal semantics. In David A. Turner, editor, *Research Topics in Functional Programming.* Addison Wesley, 1990.

[40] Simon Thompson. Lawful functions and program verification in Miranda. *Science of Computer Programming*, 13(1):181–218, 1990.

[41] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.

[42] Simon Thompson. Formulating Haskell. In *Workshop on Functional Programming, Ayr, 1992*, Workshops in Computing. Springer Verlag, 1993.

[43] Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7, 1995.

[44] Simon Thompson. Programming language semantics using Miranda. Technical Report 9-95, Computing Laboratory, University of Kent at Canterbury, 1995.

[45] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison Wesley, second edition, 1999.

[46] Simon Thompson and Claus Reinke. A Case Study in Refactoring Functional Programs. In *Brazilian Symposium on Programming Languages*, 2003.

[47] M.B. van der Zwaag. Time-stamped actions in mCRL algebras. Technical Report SEN-R0002, CWI, Amsterdam, 2000.

[48] A.G. Wouters. Manual for the mCRL tool set. Technical Report CWI Research Report SEN-R0130, CWI, The Netherlands, 2001.

Part 3 - Diagrammatic Project Plan

Outcomes:

Workpackage 1:

Workpackage 2:

Workpackage 3:

Workpackage 4:

Workpackage 5:

Workpackage 6:

Workpackage 7:

Workpackage 8:

Workpackage 9:

Workpackage 10:

Workpackage 11:

Workpackage 12: Dissemination and collaboration
This includes the PhD student writing up his/her dissertation in addition to the standard dissemination activities outlined above.

Project management:

The proposers have a long history of working together successfully for many years. The two named RAs, Clara Benac Earle and Huiqing Li, are currently being supervised by Derrick and Thompson, respectively. John Derrick worked at Kent for 15 years and will remain as a Visiting Professor after his move to Sheffield in January 2005. Their combined track record, expertese make the four staff an experienced and suitable team to carry out this work. The split of work of the staff will be as follows:

Benac Earle and Derrick: Workpackages 1, 2, 6, 7, 8, 9, 10
PhD student and Derrick: Workpackages 10, 11 (and be part of but not expected to contribute strongly to 1 and 2)
Li and Thompson: Workpackages 1, 2, 3, 4, 5, 9, 10.

Formal project management will be by 6-monthly review meetings, and these will involve our industrial collaborators T-mobile and Erlang Consulting, and will also be open to other interested parties. It will also be possible for staff to work at one site for short periods of time on a particular workpackage if that is felt desirable.