

Property-based testing - The ProTest project

John Derrick¹ and Neil Walkinshaw¹ and Thomas Arts² and Francesco Cesarini³ and Lars-Ake Fredlund⁴ and Victor Gulias⁵ and John Hughes⁶ and Simon Thompson⁷

¹Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK

²Goteborgs Universitet, Goeteboug, Sweden

³Erlang Training and Consulting LTD, London

Universidad Politecnica De Madrid, Madrid, Spain

⁴Lambdastream Servicios Interactivos SL, A Coruna, Spain

⁶Quviq AB, Savedalen, Sweden

⁷University of Canterbury, Canterbury, Kent, UK

Email: J.Derrick@dcs.shef.ac.uk

Abstract. The ProTest project is an FP7 STREP on property based testing. The purpose of the project is to develop software engineering approaches to improve reliability of service-oriented networks; support fault-finding and diagnosis based on specified properties of the system. And to do so we will build automated tools that will generate and run tests, monitor execution at run-time, and log events for analysis.

The Erlang / Open Telecom Platform has been chosen as our initial implementation vehicle due to its robustness and reliability within the telecoms sector. It is noted for its success in the ATM telecoms switches by Ericsson, one of the project partners. In this paper we provide an overview of the project goals, as well as detailing initial progress in developing property based testing techniques and tools for the concurrent functional programming language Erlang.

1 Introduction

Communication networks, based on telephony, wireless and Internet, have over the last few years been converging. At the present time and for the foreseeable future, more and more services will be added to these merging networks. Moreover, these services are becoming more complex, both in themselves and in their interactions with each other and their end users. The telecoms industry has an admirable record in providing reliability and robust services to its clients, and indeed it is the telecoms industry that can point to 5-nines reliability: that is 99.999% reliability, of their core systems.

This context provides the motivation of the ProTest project - namely that of maintaining 5-nines reliability in future service-oriented networks and systems.

The software for new services and network devices is rapidly growing in complexity, among other things because of the variety of formats and multiplicity of delivery modes evident in modern communication protocols (with thousands

of optional fields, for instance). In addition, such software needs to be context-aware, since the requirements vary when the same software is used in different ways. There are several ingredients for ensuring that such complex systems provide the expected reliability, among them choosing a good architecture, using the right technologies, improving the software process, and also being extremely thorough and efficient in *testing*.

Testing of complex systems is difficult and time-consuming in the extreme, and in the ProTest project we build upon the innovative idea of using properties as objects for testing software. In order to deliver dynamic services and interoperable network applications with guaranteed properties, we focus testing around these properties.

The economic motivator is that testing with properties as objects improves the competitiveness of software developers, since they can deliver higher quality software for a lower price. It also allows collaborating companies to improve the definition of their software interfaces and therewith improve the compatibility between their services.

Our objective is to deliver methods and tools to support property-based development of systems, and in order to do so we need tools to integrate property-based testing into the development life cycle. To this extent we are conducting work along four technical themes as follows:

Property discovery. Current testing is based on sets of test cases embedded in test suites; over the lifetime of the project we will aim to provide tools to aid the software developers to extract properties from this test data. Current specifications and models are often informal: so we will develop specialised property languages to ease the formalisation of existing specifications.

Test and property evolution. All software systems are subject to change and evolution; we will thus provide tools to support the evolution of tests and properties in line with the evolution of the system itself.

Property monitoring. Not all properties can be tested in advance of systems being executed, and so we will provide tools to support the post hoc examination of trace details for conformance to (or indeed violation of) particular constraints.

Analysing concurrent systems. At the heart of service oriented systems is concurrency: servers will provide services to multiple clients in a seamlessly concurrent way; services will federate to provide complex functionality through concurrently performing parts of a task. We will provide tools by which such concurrent systems can be analysed for fundamental properties by way of model-checking and testing.

In subsequent sections of this paper we explain work in progress under each of these themes.

2 Background

The ProTest project aims to introduce property-driven development into the software engineering process. Property-driven development can be used in a variety of programming languages and systems. The particular platform chosen for

initial implementation of the project is Erlang/OTP (Open Telecom Platform), but a crucial aspect of our proposal is the dissemination and adoption of the approach much more widely, particularly into the model driven development arena (UML) and other implementation languages (C/C++, Java, etc).

Erlang/OTP has been chosen as the implementation vehicle because of its robustness and reliability within the telecoms sector; witness, for example, its success in the implementation of the AXD301 ATM telecoms switch by Ericsson, one of the project partners. Erlang [AVWW96] is a concurrent functional language with specific support for the development of distributed, fault-tolerant systems with soft real-time requirements. Language and implementation design have aimed from the start to support a concurrency-oriented programming paradigm and the massively concurrent systems that it leads to.

The project consortium contains a balance of academics from Universities of Sheffield, Kent, Politecnica de Madrid, Goteborg, Chalmers University of Technology, SMEs, and a larger company. One of the SMEs is Quviq which is a spin-off from academia, founded to commercialise the property-based testing tool QuickCheck. The remaining industrial partners are system builders (Ericsson, LambdaStream), consultants, and trainers (ETC), who provide invaluable insights into what is required of practical tools, what properties will need to be checked, and ways of fitting the results from the project into practical software development methods.

Our own work on QuickCheck [AHJW06] combines random test case generation, with a flexible language for specifying generators, with the use of properties to adjudge success [CH00]. The inevitable noise in random test cases is removed by automatic simplification, using an approach resembling Zellers delta-debugging [ZH02]. This technique enabled us to isolate subtle faults in industrial telecommunications software [AHJW06], and has also been used successfully to test software for space missions [GHJ07].

Refactoring has become a well-known technique, particular in the realm of object oriented software development. It is standard for Integrated Development Environments, such as Eclipse, NetBeans and IntelliJ IDEA, to support a selection of refactorings, particularly those to do with the structure of the code base. Refactorings are also commonly discussed in the context of transforming code so that it conforms to a particular design pattern or coding standard. Here we build on existing work undertaken at Kent who have developed refactoring tool support for functional programming [Tho04] in the languages Erlang [LT08] and Haskell [LTR05] and their relationship [LT06].

Trace analysis is a natural extension to testing. Instead of only studying the outcome of a test case, all events (at some appropriate level of detail) during the test execution are recorded in a trace. By analyzing the trace in an intelligent way more information can be extracted from a single test. The Erlang run-time system has a built-in trace recording functionality, which has lead to wide-spread use of trace analysis as a verification technique for Erlang systems. Trace analysis for Erlang systems has been studied by [AF02] and further by [ACS04]. Our previous work on trace analysis for Erlang includes trace abstraction, in which

an approximation of a system's state space is built from an actual concrete trace. This is done using an abstraction function; the resulting state space is called an abstract trace.

Model checking offers the promise of a push-button solution for verification, and during the last twenty years many researchers have been pursuing that goal. In practise the technique still suffers from the well known state explosion problem, i.e. models become too large for analysis. Thus a priority is developing tractable models by abstracting from the full complexity of the artefact being verified. Our work in the project on model checking will investigate the integration of property-based testing and model checking techniques for Erlang. As model checking inevitably fails to fully verify a piece of software (e.g., due to state explosion or the problem of constructing an accurate model from a complex program), we have to resort to testing. But, in fact, testing and model checking are often complementary techniques. In ProTest we will explore their combination in model based testing (to provide accurate estimations of space coverage, to provide a test oracle, etc) and to explore non-exhaustive model checking as an alternative to testing for highly concurrent and complex distributed systems.

3 Property discovery

Our work on property discovery covers two main aspects, one dealing with obtaining properties from a specification, the other dealing with obtaining properties from a library of existing test cases.

3.1 Properties from specifications

To enhance how QuickCheck can be applied to other languages, we have produced a library for testing finite state machines, which has been used in an industrial project in which the UML design tool Rose/RT was connected to QuickCheck, allowing systems designed in Rose/RT to be using QuickChecks finite state machine library. We have also developed a general approach to test C software with QuickCheck. In this way, all QuickCheck libraries developed in the ProTest project also become available to Rose/RT and C programmers.

We have also developed two ways of obtaining properties from specification, viz. obtaining properties from data type definitions and from databases. The methods have been evaluated in a number of industrial projects, and some subtle errors were identified in the financial systems of these companies and the methods proved useful [ACH08]. The novelty here is that one is assured that the properties together span the complete set of all possible tests.

Going further we have developed a fully automatic method to generate properties from purely functional descriptions for both Haskell and Erlang. This tool, called QuickSpec [Hug08], can automatically generate properties for a given library of functions. QuickSpec reads in an API of an Erlang module or a Haskell module, and automatically produces a list of equations that hold for the functions in that module. The method uses random testing to do this (no heavy

theorem proving is performed); the only extra input the tool might require is some information on how to generate test data.

For example, given the function names of the standard list functions `append` (`++`), `reverse`, `tail`, `cons`, empty list (`[]`), `insert` and `sort`, the tool produces the following algebraic properties of the functions, fully automatically, in about 1 second:

```

1: insert(X, []) = [X]
2: insert(X, [X|Xs]) = [X| [X|Xs]]
3: insert(Y, [X]) = insert(X, [Y])
4: insert(Y, insert(X, Xs)) = insert(X, insert(Y, Xs))
5: reverse([]) = []
6: reverse([X]) = [X]
7: reverse(reverse(Xs)) = Xs
8: sort([]) = []
9: sort([X|Xs]) = insert(X, sort(Xs))
10: sort(insert(X, Xs)) = insert(X, sort(Xs))
11: sort(reverse(Xs)) = sort(Xs)
12: sort(sort(Xs)) = sort(Xs)
13: sort(Ys++Xs) = sort(Xs++Ys)
14: stail([]) = []
15: stail([X|Xs]) = Xs
16: Xs++[] = Xs
17: []++Xs = Xs
18: [X|Xs]++Ys = [X|Xs++Ys]
19: reverse(Xs)++[X] = reverse([X|Xs])
20: reverse(Xs)++reverse(Ys) = reverse(Ys++Xs)
21: stail(Xs)++Xs = stail(Xs++Xs)
22: (Xs++Ys)++Zs = Xs++(Ys++Zs)

```

The basic method we use is the following. We start by generating a finite set of well-typed terms that contain variables (in the above example there are 2298 such terms of depth 3). Next, we compute equivalence classes of these terms, by means of random testing and refining: we start by assuming that all terms are in the same equivalence class, and partition equivalence classes into smaller ones by running random tests and inspecting the values of the terms (in the above example, this results in 1931 equivalence classes). For each equivalence class, we pick one representative, and produce equations between that representative and all other terms in an equivalence class. For the example, this results in 367 equations, these are all equations that are true, but there are clearly too many to be useful, thus we spent some effort into producing a list of non-overlapping algebraic equations.

When one naively generates equations that hold between terms, many of which are not independent. To reduce the number, we have developed several filtering algorithms that remove superfluous equations. Choosing the right filtering algorithm constitutes finding a balance between (1) not keeping too many

equations, (2) how expensive is it to check that equations follow from other equations, (3) not removing too many equations (even though an equation follows from other ones, it might still be useful to have in the list). The algorithm we finally settled for uses a congruence closure algorithm to approximate if an equation follows from a set of equations.

We have applied QuickSpec to a number of concrete Erlang and Haskell modules. Most notably, we applied it on the Erlang standard functional array library, and on a library for fixed-point arithmetic that was written by a company in South Africa. Exploring the properties that QuickSpec produced (and the properties it did not produce!) was a great way of understanding code that someone else had written, and has lead us to come up with a number of concrete techniques that may be used for applying QuickSpec in this way. Other applications of QuickSpec include providing a cheap and easy way for programmers and testers to start writing properties.

3.2 Reverse Engineering

We have developed two methods to extract properties from test cases - one dynamic, the other static. That is, in the first approach, the test cases are run, generating traces for the program. From these traces a finite state machine can be abstracted. This is described fully in the companion paper [WD09] as well as in [WDG09].

The second approach works on the level of the source code of the test cases. It is a guided automatic approach; testers know best what part of the test case they like to generate and what part they want to keep specific. Recent work with test suites from Ericsson, and with tests from an Open Source project (Engineyard's Natter application) confirm that this approach is a fruitful one.

3.3 Building Domain Specific Languages

QuickCheck has long provided a DSL (Domain Specific Language) for specifications based on abstract state machines; however, this DSL represents states as arbitrary Erlang data values - for example, a list of key-value pairs if modeling a key-value store. With this approach, each operation of the API under test is applicable in every state, unless an explicit precondition is given to restrict this. Software is commonly specified instead via a state transition diagram, in which states are distinguished by name, and operations are typically applicable only in a certain named state.

Of course, such a specification can be based on the previous state machine library, but doing so in effect encodes the structure of the diagram in an ugly way in many different places in the code.

To help overcome this, we have developed a new FSM library, implemented on top of the original one, which separates state names and state data. Specifications using the new library are much more concise and perspicuous than the equivalent specification using the old one. Our FSM library allows weights to be assigned to transitions, but assigning weights well is difficult, since changing a weight

on one transition can affect the execution frequency of many others in quite non-obvious ways. We have therefore developed an optimization criterion for weighting (which essentially tries to distribute test effort as evenly as possible across the transitions in the state diagram), and an approximation algorithm for assigning weights automatically. Although the algorithm is not optimal (and finding an optimal solution appears to be NP-hard), it usually produces good results.

The weight assignment algorithm can take priorities into account - for example, if the user specified that testing the lock transaction is 10x as important as testing others (for example, because it contains new code), then the weight assignment algorithm results in the distribution to the left. Note that unlock is also assigned a higher weight necessary, since without an unlock, we can never perform more than one lock in a test case. The new library has now been released as a part of Quviqs product.

4 Refactoring

Our second strand of work addresses software evolution and the way that this impacts on testing, and in particular property-based testing. The Wrangler refactoring tool [LT08,ST08], developed at Kent, is used to support refactorings of tests, test-aware refactorings and property discovery.

Initial work has investigated the impact of various refactorings on testing as practised in three systems:

- EUnit (for unit testing of Erlang systems),
- Quviq QuickCheck (for property-based testing of Erlang systems), and
- Common Test / OPT Test Server (for system testing)

and we describe a selection of work below.

4.1 Duplicate/Similar code detection in Wrangler

Duplicated/similar code is common in software, especially in test cases. For example, in industrial test suites, some test case functions only differ in an atom and a record definition. It would be desirable to have a generalised abstraction of these similar test case functions, and make each test case an instance of the generalised abstraction.

Wrangler's support for "duplicated code detection" and "expression search" is able to report code fragments that are syntactically identical after semantic-preserving renaming of variable names, ignoring variations in literals, layout and comments.

The requirement of "syntactic identity" is somehow restrictive because it could not detect code fragments that look similar but are not syntactically identical. For instance, Wrangler's original "expression search" would not report the following two pieces of code as clones because of the slight syntactical difference in the record field "codec" though they look very similar.

```

Code fragment 1:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?COMMENT("Test case create_2 started.",[]),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SidMux = {mux_id_1, h223_id_1},
{TdmSid, LocalData, _, _} = precondition_one_blade_tdm_mux_create(SidMux),
?CHECK(ok, hcfTraceServerSupport, start, [{brchDspRhI, exported}]),
SidLc = {mux_id_1, audio_id_1},
CreateData = #brchMuxLcAccess{sid = SidLc,
stream_type = ?BRCH_AUDIO,
local_data = LocalData,
codec = {?AMR,
{?R_122, ?BRCH_DISABLED,
?BRCH_DISABLED, ?BRCH_BIT},
33, 44, 40},
event_module = iptermCb},
?CH(1, brchShI, create, [[CreateData]]),
?CHECK([], hcfTraceServerSupport, get_trace_list, []),
clean_up([SidLc, SidMux, TdmSid]),
?RESULT("DONE", []).

```

```

Code fragment 2:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?COMMENT("Test case create_3 started.",[]),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SidMux = {mux_id_1, h223_id_1},
{TdmSid, LocalData, _, _} = precondition_one_blade_tdm_mux_create(SidMux),
?CHECK(ok, hcfTraceServerSupport, start, [{brchDspRhI, exported}]),
SidLc = {mux_id_1, audio_id_1},
CreateData = #brchMuxLcAccess{sid = SidLc,
stream_type = ?BRCH_AUDIO,
local_data = LocalData,
codec = {?G723_1, {?R_53, ?BRCH_DISABLED},
33, 44, 40},
event_module = iptermCb},
[{ok, [{SidLc, _IntCep}], ?BRCH_REPLICATION_NEEDED}] =
?CH(1, brchShI, create, [[CreateData]]),
?CHECK([], hcfTraceServerSupport, get_trace_list, []),
clean_up([SidLc, SidMux, TdmSid]),
?RESULT("DONE", []).

```

To be able to detect this kind of similarity, we have extended Wrangler with a "Similar expression search". The functionality allows the user to search for expressions that are similar to the expression selected according to a similarity score specified by the user.

Furthermore, "Similar expression search" also automatically generates the least general common abstraction of those similar expressions found, which is also known as anti-unifier. With the example above, Wrangler would suggest the generalised abstraction as:

```
new_fun(NewVar_1, NewVar_2, NewVar_3) ->
?COMMENT(NewVar_1, []),
SidMux = {mux_id_1, h223_id_1},
{TdmSid, LocalData, _, _} = precondition_one_blade_tdm_mux_create(SidMux),
?CHECK(ok, hcfTraceServerSupport, start, [{brchDspRhI, exported}]),
SidLc = {mux_id_1, audio_id_1},
CreateData = #brchMuxLcAccess{sid=SidLc,
stream_type = ?BRCH_AUDIO,
local_data=LocalData,
codec={NewVar_2, NewVar_3, 33, 44, 40},
event_module=iptermCb},
[{ok, [{SidLc, _IntCep}], ?BRCH_REPLICATION_NEEDED}] =
?CH(1, brchShI, create, [[CreateData]]),
?CHECK([], hcfTraceServerSupport, get_trace_list, []),
clean_up([SidLc, SidMux, TdmSid]),
?RESULT("DONE", []).
```

Being able to generate the least general abstraction automatically speeds up the similar code elimination process, because the user does not need to inspect the differences manually, and generalise the function step by step.

The notion of least general abstraction (anti-unifier) and the definition of "similarity" need to be refined further, but for the moment we have a working definition. We are now also in the process of designing a more efficient algorithm so that we could apply "similar code detection" to large projects, as well as investigating a more general notion of "similarity" than having a non-trivial common generalisation.

Apart from the work on duplicate code detection and the introduction of "similar expression search", a number of new refactorings have also been added including the introduction macros (optionally with parameters), folding expressions against a macro definition, and the normalisation of record expressions.

4.2 Extension to Wrangler to refactor EUnit test data

We have been working on the extension of the Wrangler tool to accompany the basic refactorings in Wrangler with refactorings of EUnit test data. This extension has two aspects:

1. When application code is refactored, Wrangler should make sure that the test code of the application code is also refactored consistently.
2. Since test code is also Erlang code, it can be refactored in its own right, but Wrangler needs to make sure the refactoring of test code preserves the test framework's particular idioms, such as naming conventions.

The extension affects all the refactorings that change module/function/macro interfaces, such as renaming, generalisation, move function between modules, function extraction, etc.

The major challenge with extending Wrangler to the EUnit test framework lies in the interpretation of symbolic representation of test data, and the multiple roles of atoms in the Erlang language. For example, with EUnit's test data representation, a single module name, which is an atom, can be used to represent the whole test set from the exported test functions of the named module; so when the named module is renamed, Wrangler needs to make sure all the related uses of this module name in the test data are renamed, and also make sure that atoms with the same name, but not used as a module name are not renamed.

To ensure that Wrangler refactors test data correctly, we designed some invariants which should hold for a refactoring. For example, for each test generation function, *F* say, affected by a renaming refactoring, suppose *F* becomes *F'* after the refactoring, then the following invariant should hold:

```
rename(parse(F())) == parse(F'())
```

where function *parse* transforms the test set representation into a normal form, *rename* does renaming in the normal form in which each atom's role can be decided precisely. If the above invariant does not hold for a particular test generation function, Wrangler will ask the user for manual inspection.

4.3 Wrangler and Eclipse: integration with Erlide

Another strand of work has been to support the integration of Wrangler into the Eclipse binding for Erlang, Erlide. Erlide is under active development at Ericsson, as well as being made available freely to the Erlang community. Wrangler is currently a part of the standard Erlide distribution, freely available for download.

Integration with Eclipse through Erlide provides a number of advantages over emacs. For example, it has a well-defined notion of project, and so this gives a scope to refactorings which affect more than one module; it has a well-defined distribution and update mechanism, which means that users will automatically pick up the latest version of the tool (should they choose to); it provides multiple views of a code base, so that users can access refactorings in different ways. In addition, through its refactoring API, it provides some facilities "for free" such as preview of the effect of refactorings (across multiple modules), and through its interface it is possible to present results of searches or the effect of a multisite refactoring in a more explicit way than emacs. For instance, search results can be browsed, and choices for multi-site refactorings be specified through a series of check boxes.

Future work will see the creation of a new integration structure which relies more on Erlide. With this development it will be possible to access the refactorings through the Outline, Navigator and Duplicated Code views as well as

through the Refactor menu at present. This will in turn simplify the User Interface, and eliminate a number of current error possibilities which arise as a consequence of the form of the interface.

5 Property monitoring

The final goal of the audit-log analysis is to be able to monitor audit-log properties at (or close to) real time, and to do so will require a way to rigidly specify what should be checked. Thus, in parallel to the experiments with an early prototype, we have studied the problem of describing inter-log-file relations.

Work on developing a prototype monitoring tool has concentrated on an initial prototype that automatically analyses a (set of) log file(s), given a description of what constitutes the 'key' and what is the interesting 'value'. This simple analysis can for example track a session ID through several separate log files, or track a single request by focusing on a request ID. Events are either sorted by appearance in the log files or by their timestamps. The sequences are presented graphically using the graphviz visualization package. The prototype is also able to check sessions against a specification (represented as a state machine). A difficulty identified with this prototype is that we need to have a more flexible way of describing parts of log entries that are interesting and how these should be used, and this will be part of our future plans.

Use of the prototype showed that we need to design the logs in a system in such a way that we can define and check properties easily. We have been able to use a simple example of an SMS Log System to design the criteria and guidelines defined earlier to provide a testing basis for the tool.

6 Analysing concurrent systems

Our work on support for concurrent system analysis has included a number of themes. Part of this is working out how to shrink counter-examples resulting from an error found in a system, and to support repeatable testing. Another major theme is development of McErlang, a model-checker for Erlang.

6.1 Shrinking Trace counter-examples

The goal here is to investigate, and implement, methods to shrink trace counter-examples (resulting from testing or model checking concurrent systems) to ease the task of understanding the reason for a fault detected during testing. As such trace counter-examples frequently grow very large, having such a reduction facility is highly desirable. Our work has resulted in the development of a new tool, PULSE, which is now implemented as part of the commercial QuickCheck distribution. PULSE has been used for finding race conditions in industrial software. See [CPS⁺09] for more information.

To achieve property-based testing of concurrent software, several challenges have to be overcome. We must be able to decide whether tests have passed, and to run tests repeatably.

Our approach to automatically simplifying failing tests is based on running many simpler variations on the first failing test found, culminating in a minimal example that provokes a fault in the software under test. Finding such minimal failing tests is invaluable in speeding fault diagnosis. Yet our approach depends fundamentally on being able to repeat a test, with the same result as the first time it was run; finding minimal failing tests then requires that we can repeat smaller tests, in the same way as the original failing test was run. In concurrent programs, where the scheduling can vary from run to run, achieving repeatable behaviour is already a challenge.

Our first goal is thus to enable repeatable testing of concurrent Erlang code. This could be achieved by modifying the underlying Erlang virtual machine to use a custom, controllable process scheduler. But in practice, users will not be interested in using a custom version of the Erlang VM to test their systems - in fact, many projects continue to use outdated virtual machines long after upgrades are released, to avoid problems in their own software that might be caused by changes in the behaviour of the VM. Thus we consider it essential to achieve repeatable testing without changing the underlying Erlang VM. As multicore systems become more and more prevalent, it will be less and less reasonable to assume that the underlying scheduler can be replaced.

Our approach is instead to instrument the code under test, to make it communicate with a scheduler of our own design, written in Erlang, such that our scheduler can impose purely deterministic execution on the code under test, regardless of the underlying concurrent execution. We have developed an instrumenting compiler (in only 400LOC) which handles almost full Erlang, and an associated scheduler which takes control of the order of delivery of interprocess messages. By varying this order, we can even test the behaviour of distributed systems (which have a different semantics for message passing) on a single Erlang node. In addition we created a way to visualize the scheduling of events, such that the analysis of error cases becomes much easier. The scheduler currently makes random scheduling choices, and has proven quite effective in revealing race conditions in the examples studied so far.

6.2 Developing model-checking techniques for Erlang

The other strand of work in our support for concurrency involves development of model checking as a complementary verification technique to the use of testing. Our initial goal was to deliver a model checker that supports a very large fragment of the Erlang language (e.g., with full support for all Erlang data types, the distributed Erlang API, and many OTP behaviours) to ease the task of constructing a verifiable model from an Erlang program.

A prototype model checker existed at the start of the project, and we have concentrated on delivering a number of enhancements to it, including:

- support for model checking a much larger language fragment. To achieve this a new source-to-source translation was realised as a number of transformations on HiPE Core Erlang code – an intermediate code level in the Erlang compiler. In addition more OTP behaviours are handled (`gen_fsm`, `gen_event`, partial support for ets tables, ...). In fact we are able to use the source code for some Erlang/OTP modules, without changes, in model checking.
- the implementation of an alternative small-step Erlang semantics which is able to detect more program errors, but which may yield substantially larger state spaces,
- initial support for using multiple processors (SMP) for model checking,
- improved handling of Linear Temporal Logic claims through the integration of a new translator from Linear Temporal Logic to Buchi automata (see discussion below),
- support for combining simulation and model checking algorithms to reduce the state space needed to verify a program. This is used to reduce the cost of using OTP behaviours such as e.g. the supervisor behaviour,
- providing user documentation, including a tutorial, a user manual, and a web page.

This has resulted in the production of the McErlang model checker which has been released as open source under the agreed project license (a BSD variant); more documentation and the option to download it is available at the tool web site: <https://babel.ls.fi.upm.es/trac/McErlang/>.

A sign of the increasing maturity of the tool is that we were able to analyse a RoboCup simulation league team programmed in Erlang (comprising some 8500 lines of Erlang code) using the McErlang tool, see [EFIL08]. A number of recent improvements to the McErlang tool realised in the ProTest project are described in [EF09].

LTL-to-Buchi translation One of the additions made to McErlang during the ProTest project was to add the possibility of expressing and checking correctness properties expressed in Linear Temporal Logic (LTL). This is fairly straightforward, since LTL expressions can be automatically translated into Buchi automata. However, for model checking to be efficient it is important to produce as small an automaton as possible, thus a good translator was needed. The obvious solution was to use an existing implementation. However, this was not done for two reasons: by developing an in-house translator we avoided licensing problems (our in-house translator is licensed under the same BSD license as McErlang unlike, e.g., the LTL2Buchi translator used in the JavaPathfinder project), and secondly its proper integration into the McErlang verification framework enabled a better end-user experience (e.g., with regards to formula parsing/deparsing, conversion to an executable Erlang module, and so on).

The LTL-to-Buchi translator we have developed [Sve09] consists of the following three parts: - A rewrite engine, which aims to simplify the LTL formula. It uses a fixed set of (heuristically chosen) rewrite rules. - A core translation algorithm Construction of the Buchi automaton from the re-written LTL formula.

We use a tableau-based algorithm. - A reduction step, where optimizations such as simulation reductions and removal of non-reachable and non-accepting states, are applied to the Buchi automaton.

The efficiency of the LTL-to-Buchi translator was evaluated against two reference implementations; the LTL2Buchi translator in the JavaPathfinder and the Wring tool. Our translator clearly outperforms Wring; moreover the evaluation also uncovered a few remaining errors in the Wring tool. The resulting automata generated by our tool and LTL2Buchi are very similar in size, perhaps not very surprising since similar translation algorithms are used. However, on average our implementation generates about 1% smaller automata, when tested on randomly generated LTL formulas.

The development process (the implementation was carried out using property driven development supported by the QuickCheck tool) for the LTL-to-Buchi translator, as well as the implementation and the result of the evaluation are described in [EF09].

7 Tool integration

In addition to work on the individual tools and methods described above, we aim to integrate the tools we are building in a number of ways. As a first step, we focused on the verification of the global process registry with an approach that combined QuickCheck and McErlang. Here, the QuickCheck tool was used to generate a number of test sequences for the global process registry; these were then fed to McErlang which explored all possible interleavings of the test sequences using its model checking algorithms. Finally the results (a set of sequences of return values of a set of API calls) were checked using the QuickCheck tool. Early results are promising, as the combined tool set was also able to discover race conditions in the global process registry.

We are also working on a integration of the other relevant tools. Essentially we want to be able to run a set of QuickCheck tests where the program under test is capable of being controlled by different schedulers: (i) either using the standard Erlang program scheduler, or (ii) using the PULSE scheduler which offers more control over scheduling and a more random behaviour, or (iii) the program is controlled by the McErlang model checker which in theory can fully explore the state space corresponding to any given test case.

As an example of how the tools and methodologies can be integrated consider the following example, where we describe how we can refactor a test suite into properties.

7.1 Example

The test suite in this example is 2228 lines of code, containing 4 groups of test cases:

- 5 test cases in the create group,

- 4 test cases in the set_topology group,
- 11 test cases in the modify group,
- 10 test cases in the delete group.

It is clear that certain test cases have some similarity. For example, we have a number of occurrences where a test case for audio is repeated for video. There are two ways in which we can work with this in the refactoring tool: We can search for expressions identical to this, or we can perform a general search for code clones in the existing file (or indeed in a complete project). Currently under development is a facility to search for "similar" code.

Using this approach we automatically find that test cases create_2, create_3 and create_4 only differ in an atom and a record definition. The test cases create_2 and create_3 are for audio, the test case create_4 is for video.

```
create_2(id) -> "create_2";
create_2(doc) -> "Create basic VIG MUX + audio LC AMR segment";
create_2(setupimg) -> "";
create_2(fts) ->
"/vobs/mgwblade/HCF/HCF_CRA1190072/test/doc/15241/XYZ_FTS.fm";
create_2(class) -> auto;
create_2(time) -> {{00,00,00},{00,00,00}};
create_2(config) -> [];
create_2(main) ->
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
?COMMENT("Test case create_2 started.",[]),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
SidMux = {mux_id_1, h223_id_1},
{TdmSid, LocalData, _, _} = precondition_one_blade_tdm_mux_create(SidMux),
?CHECK(ok, hcfTraceServerSupport, start, [[{brchDspRhI, exported}]]),
SidLc = {mux_id_1, audio_id_1},
CreateData = #brchMuxLcAccess{sid = SidLc,
stream_type = ?BRCH_AUDIO,
local_data = LocalData,
codec = {?AMR,
{?R_122, ?BRCH_DISABLED,
?BRCH_DISABLED, ?BRCH_BIT},
33, 44, 40},
event_module = iptermCb},
[{ok, [{SidLc, _IntCep}], ?BRCH_REPLICATION_NEEDED}] =
?CH(1, brchShI, create, [[CreateData]]),
?CHECK([], hcfTraceServerSupport, get_trace_list, []),
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Clean up this test case
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clean_up([SidLc, SidMux, TdmSid]),
?RESULT("DONE", []).
```

If we select the body of the `create_2(main)` clause and search for expressions (i.e., similar code), we will find `create_4`, but also a `create_3`, which is also for audio, but which differs much less. We found out that generalizing code further apart from each other will result in being able to automatically include code closer to the original copy.

Using the facilities in Wrangler to generate this, we automatically get the most general antiunifier of the code, that is, variables replace subterms that are different. The most general part is copied into a new function `create_234`, since it combines the 3 test cases `create_2`, `create_3` and `create_4`.

Another refactoring ("folding") lets us now replace the bodies of `create_2`, `create_3` and `create_4` to function calls to `create_234` with different arguments.

So far, this is pure refactoring, the code that we produce has the same semantics as the original test cases. Now we introduce a step that helps us to lift test cases to properties. We collect all calls to `create_234` in one generator that randomly selects one of the alternatives and we create a property that does test each of these 3 alternatives:

```
create_234_gen() ->
oneof([audio_id_1,?BRCH_AUDIO,{?G723_1, {?R_53, ?BRCH_DISABLED}, 33,
44, 40}},
{audio_id_1,?BRCH_AUDIO,{?AMR, {?R_122, ?BRCH_DISABLED,
?BRCH_DISABLED, ?BRCH_BIT}, 33, 44, 40}},
{video_id_1,?BRCH_VIDEO,{?H264, ?BRCH_NO_OPTION, 33, 44, 40}}
]).

prop_create_234() ->
?FORALL({Media,Channel,Codec},create_234_gen(),
create_234(Media,Channel,Codec)).
```

The property should be a bit more complex, since it should return true or false, not the result of `create_234`, but for reasons of clarity we keep it simple here.

Now normal refactoring steps should be used to refactor the generator in this property to more detailed generators. We know how to do this manually, and automation will be possible, and is in our future plans.

The result will be:

```
media() ->
oneof([audio_id_1,video_id_1]).

streamtype(audio_id_1) ->
?BRCH_AUDIO;

streamtype(video_id_1) ->
?BRCH_VIDEO.
```



```

codec(audio_id_1) ->
oneof([ {?G723_1, {?R_53, ?BRCH_DISABLED}, 33, 44, 40},
{?AMR, {?R_122, ?BRCH_DISABLED, ?BRCH_DISABLED, ?BRCH_BIT}, 33,
44, 40}
]);

codec(viedo_id_1) ->
{?H264, ?BRCH_NO_OPTION, 33, 44, 40}

```

This requires that we know that there is a dependency between the different fields and it also requires automatic refactoring of the property as soon as the generators are refined:

```

prop_create_234() ->
?FORALL(Media,media(),
?FORALL(Channel,streamtype(Media),
?FORALL(Codec,codec(Media),
create_234(Media,Channel,Codec)).

```

Now the tester can add additional alternatives to the generator which will automatically increase the number of tests, without having to copy and paste test cases. In addition, the test code becomes more structured and readable.

This example shows how refactoring and related transformations in Wrangler can be used to support the extraction of Quick Check properties from 'free' tests. A similar approach allows Quick Check properties to be extracted from EUnit tests, and we anticipate implementing a suite of transformations supporting in the near future.

8 Conclusions

Our work on property discovery has already shown very promising results. Working with our industrial partners we are now close to having automatic support for extracting properties from test cases. In addition, we have worked on two other ways of obtaining properties from specification, viz. obtaining properties from data type definitions and from databases. The methods have been evaluated by the Swedish company Kreditor, and Ericssons OTP team, and have shown their immediate benefit. We have developed a tool, called QuickSpec that can automatically generate properties for a given library of functions.

We have developed two methods to extract properties from test cases. One is dynamic, the other static. In the first approach, the test cases are run, generating traces for the program, from these traces a finite state machine can be abstracted. The second approach works on the level of the source code of the test cases. It is a guided automatic approach; testers know best what part of the test case they like to generate and what part they want to keep specific.

Our work on test and property evolution has concentrated on the development of the Wrangler refactoring tool that can be used to support refactorings

of tests, test-aware refactorings and property discovery. We have investigated the impact of various refactorings on the industrial practise of testing using: EUnit, QuickCheck, and Common Test / OPT Test Server, and have worked on extending Wrangler with refactorings of EUnit test data. We have begun the integration of Wrangler into the Eclipse binding for Erlang, Erlide.

In property monitoring we have developed a prototype tool that automatically analyses a set of log files given a description of what constitutes the 'key' and what is the interesting 'value', and is capable of handling some non-trivial inter-log-file relations.

We have made significant progress in our work on analysing concurrent systems. Our work on developing methods to shrink trace counter-examples (resulting from testing or model checking concurrent systems) has resulted in the development of a new tool, PULSE, which is now implemented as part of the commercial QuickCheck distribution. PULSE has been successfully used to find race conditions in software provided by an industrial partner.

In addition we have developed a model checker, McErlang, which was released as open source under the agreed project license that supports a very large fragment of the Erlang language to ease the task of constructing a verifiable model from an Erlang program.

References

- [ACH08] Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang data types with Quviq QuickCheck. In Teoh and Horváth [TH08], pages 1–8.
- [ACS04] Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in erlang. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2004.
- [AF02] Thomas Arts and Lars-Åke Fredlund. Trace analysis of erlang programs. In Rex L. Page and John Hughes, editors, *Erlang Workshop*, pages 16–23. ACM, 2002.
- [AHJW06] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq Quickcheck. In Marc Feeley and Philip W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang (Erlang’06)*, pages 02–10. ACM Press, 2006.
- [AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
- [CH00] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
- [CPS⁺09] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 149–160. ACM, 2009.
- [EF09] Clara Benac Earle and Lars-Åke Fredlund. Recent improvements to the mcerlang model checker. In Earle and Thompson [ET09], pages 93–100.
- [EFIL08] Clara Benac Earle, Lars-Åke Fredlund, José Antonio Iglesias, and Agapito Ledezma. Verifying robocup teams. In Doron Peled and Michael

- Wooldridge, editors, *MoChArt*, volume 5348 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2008.
- [ET09] Clara Benac Earle and Simon J. Thompson, editors. *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang, Edinburgh, Scotland, UK, September 5, 2009*. ACM, 2009.
- [GdM08] Robert Glück and Oege de Moor, editors. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. ACM, 2008.
- [GHJ07] Alex Groce, Gerard J. Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE*, pages 621–631. IEEE Computer Society, 2007.
- [Hug08] John Hughes. Formal Specification for Free! In Teoh and Horváth [TH08].
- [LT06] Huiqing Li and Simon Thompson. Comparative study of refactoring haskell and erlang programs. In *SCAM*, pages 197–206. IEEE Computer Society, 2006.
- [LT08] Huiqing Li and Simon J. Thompson. Tool support for refactoring functional programs. In Glück and de Moor [GdM08], pages 199–203.
- [LTR05] Huiqing Li, Simon Thompson, and Claus Reinke. The haskell refactorer, hare, and its api. *Electr. Notes Theor. Comput. Sci.*, 141(4):29–34, 2005.
- [ST08] Nik Sultana and Simon J. Thompson. Mechanical verification of refactorings. In Glück and de Moor [GdM08], pages 51–60.
- [Sve09] Hans Svensson. Implementing an ltl-to-büchi translator in erlang: a protest experience report. In Earle and Thompson [ET09], pages 63–70.
- [TH08] Soon Tee Teoh and Zoltán Horváth, editors. *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, Victoria, BC, Canada, September 27, 2008*. ACM, 2008.
- [Tho04] Simon Thompson. Refactoring functional programs. In Varmo Vene and Tarmo Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 331–357. Springer, 2004.
- [WD09] Neil Walkinshaw and John Derrick. Incrementally discovering testable specifications from program executions. In *FMCO*, 2009.
- [WDG09] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In *Formal Methods (FM’09)*, volume 5850 of *LNCS*, pages 305–320. Springer, 2009.
- [ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.