Tanti, E. and Francalanza, A. (2015). *Xjenza Online*, 3:31–35.

Xjenza Online - Journal of The Malta Chamber of Scientists
www.xjenza.org
DOI: 10.7423/XJENZA.2015.1.04

Research Article

Towards Sound Refactoring in Erlang

E. Tanti, A. Francalanza
Department of Computer Science, Faculty of ICT, University of Malta

Abstract. Erlang is an actor-based programming language used extensively for building concurrent, reactive systems that are highly available and suffer minimum downtime. Such systems are often mission critical, making system correctness vital. *Refactoring* is code restructuring that improves the code but does not change behaviour. While using automated refactoring tools is less error-prone than performing refactorings manually, automated refactoring tools still cannot guarantee that the refactoring is correct, *i.e.*, program behaviour is preserved. This leads to lack of trust in automated refactoring tools. We first survey solutions to this problem proposed in the literature. Erlang refactoring tools as commonly use approximation techniques which do not guarantee behaviour while some other works propose the use of formal methodologies. In this work we aim to develop a formal methodology for refactoring Erlang code. We study *behavioural preorders*, with a special focus on the *testing preorder* as it seems most suited to our purpose.

1 Introduction

There exist a number of tools which assist with the construction and maintenance of software. One example of such tools are *refactoring tools* which automate the process of modifying a software system in such a way that improves the internal structure of program code but at the same time does not alter the external functionality or behaviour of the code. Examples of such improvements include improved readability, such as through compliance with certain code practices, improved efficiency, reusability, extensibility and maintainability (Mens & Tourwé, 2004). Also, recently (Brown, Hammond, Danelutto & Kilpatrick, 2012, 2013), the use of refactoring tools has been proposed to automatically preform the complex task of transforming single threaded code into *concurrent* code.

The growth of multicore, and even more recently many-core, computer processors has made concurrency a crucial consideration for developers. Taking advantage of the potential speedup that can be achieved using multicore computers is one of the main driving forces for programming concurrently. However, concurrent programs are difficult for developers to *construct* and for tools to *analyse*.

There exist a number of *concurrency models* which allow developers to construct concurrent programs. The *actor model* is one such abstraction which eases the construction of concurrent programs, providing an alternative to the currently predominant *shared memory* concurrency model. Actors are suitable for structuring highly concurrent software systems which scale up to many-core processors, as well as scale out to clusters and the cloud (Cesarini & Thompson, 2009; Haller & Sommers, 2011). This has led to a steady stream of research and industrial development, contributing to a renewed interest in actors in academia (Karmani, Shali & Agha, 2009; Haller, 2012; Sutter & Larus, 2005). Examples of programming languages which have adopted the actor model include Erlang (Armstrong, 2007; Cesarini & Thompson, 2009) and Scala (Haller & Sommers, 2011). There also exist a number of actor frameworks such as Akka which run on the Java Virtual Machine (JVM) with APIs available for both Java and Scala (Haller, 2012) amd Microsoft's Asynchronous Agents Library for .NET (Karmani et al., 2009). In our work we have chosen to focus on Erlang as it is one of the most mature actor languages (Armstrong, 2010).

There exist a number of refactoring tools for actor-based programming languages such as Erlang, Tidier (Sagonas & Avgerinos, 2009) and Wrangler (Li, Thompson, Orosz & Tóth, 2008). The use of automated refactoring tools is generally faster and less likely to introduce mistakes than refactoring manually (Murphy-Hill, Parnin & Black, 2009). Despite this, refactor-

Correspondence to: E. Tanti (erica.tanti.09@um.edu.mt)

ing tools are, as discussed in a survey by Murphy-Hill (Murphy-Hill, 2007), largely underused. Amongst the reasons for this phenomenon the people surveyed cited lack of trust in automated refactoring tools. To counter this problem, most of the refactorings provided, including those by existing ERLANG refactoring tools such as Tidier (Sagonas & Avgerinos, 2009), are simple as this is the only way to ensure that behaviour is preserved. Despite this, bugs may still be found in refactoring tools, even after extensive testing (Li & Thompson, 2008; Soares, Gheyi, Serey & Massoni, 2010). Therefore tools permitting more complex refactorings such as Wrangler (Li et al., 2008) use *side-conditions* or *assertions* to attempt to ascertain whether a refactoring is valid, but do not guarantee behaviour preservation. To mitigate the problem, such tools, including Wrangler, are often *semi-automatic*, thus necessitating some human interaction and support an *undo mechanism* which Mens and Tourwé (2004) state is required when a tool cannot guarantee behaviour preservation.

In this paper, we first survey and discuss existing methodologies for determining behaviour preservation in refactoring tools in Section 2. In this section, we discuss both approximation techniques and precise formal techniques. We then discuss in Section 3 how to apply previous work to develop a formal behavioural theory for ERLANG and which may be applied to ERLANG refactoring tools. Section 4 concludes.

2 Techniques used to prevent erroneous refactorings

In this section we give an overview of techniques commonly used to prevent erroneous refactorings. We first present a brief overview of approximation techniques commonly used which do not guarantee behaviour preservation. We then discuss formal methodologies used in previous work.

2.1 Approximation Techniques

A number of approximation techniques exist for determining behaviour preservation. One relatively simple and common way to check for a refactoring's correctness is to perform conditional checks on the code. These come in the form of (1) *invariants* that should remain satisfied throughout the process, and (2) *pre-* and *postconditions* that should hold before and after the refactoring has been applied respectively. These conditions serve only to approximate correctness as only some of the set of all possible properties are checked. Additionally, statically checking certain properties may be computationally expensive or impossible. An example of such work includes Li and Thompson (2007) work checking Wrangler (Li et al., 2008) refactorings and Drienyovszky *et al.*'s work (Drienyovszky, Horpácsi & Thompson, 2010) checking Wrangler and RefactorErl refactorings, using Quviq's automated testing tool QuickCheck (Hughes, 2007).

However such methods are only approximations and thus do not supply us with definitive guarantees. In safety-critical software, undoing such a refactoring when a bug is found, and thus when the damage may already have been done, is unsatisfactory. The expense of introducing a bug, even for a short time, outweighs the possible benefits of refactoring.

2.2 Formal Methodologies

The problem of introducing bugs despite the precautions described above could potentially be avoided if there was a way to truly ensure that the refactoring is *correct*, *i.e.,* that behaviour has been preserved. To this end, there has been some previous work on the use of *formal techniques* which may be used to ensure behaviour preservation. Work carried out by Ward and Bennett (Ward & Bennett, 1995), Li and Thompson (Li & Thompson, 2005) and Sultana and Thompson (Sultana & Thompson, 2008) use a number of different formal techniques to determine whether behaviour has been preserved, similar to the goals of this work. Complementary to such techniques there are also *graph transformations* whereby software is represented as a graph, and restructurings correspond to transformation rules. Graph transformations are ideal for replacing occurrences of poor design patterns with good design patterns (Mens, Demeyer, Bois, Stenten & Gorp, 2003; Mens & Tourwé, 2004). These graph transformations must be formally proven to be correct.

2.2.1 Previous applications of formal methodologies

Ward and Bennett's (Ward & Bennett, 1995) Wide Spectrum Language (WSL) is a simple, unambiguous, formal language with pre-proven refactorings, thus guaranteeing the refactoring's correctness. Translators are then written to and from WSL to the target language. Using an intermediate language in this tool proved to be useful beyond the usual advantages of code improvement as it may be used for reverse engineering, that is to generate specifications and to translate code from one language to another. However, as their language is very general purpose, their translated code is often cumbersome and the translations themselves cannot be formally proven to be accurate.

Work (Li & Thompson, 2005) exists on proving behaviour preserving refactorings for the Haskell, which is a functional programming language like ERLANG. They formalise a core subset of Haskell using a simple lambda-calculus and a number of semantic equivalence reduction rules. Each rule is known to preserve semantic equivalence, thus a refactoring made up of a combination of these rules is also considered semantically equivalent.

However, they do not specify the equivalence: theory used (Li & Thompson, 2005).

More recently, and more similar to our ultimate goal, is the work of Sultana and Thompson (2008) who use a proof assistant Isabelle/HOL to verify refactorings for two "fragments" (Sultana & Thompson, 2008) of the programming languages Haskell and ERLANG, using preconditions and behavioural equivalence. They aimed to study the methodology for equivalence checking, which could later be extended for larger, more realistic languages and for larger refactorings. Their preliminary observations are promising.

3 Aims

Given the previous work on formal methodologies for guaranteeing behaviour preservation for refactoring tools, particularly Sultana and Thompson's (Sultana & Thompson, 2008) work, we felt that it would be appropriate to delve deeper into the topic of behavioural equivalence. To this end, we first discuss formal means of modelling the ERLANG language which may then be used to prove behavioural properties of the language

3.1 Language Formalisations

To be able to study behavioural equivalence theories for refactoring ERLANG code, we first construct a formal calculus for the language. In this section we give a review of existing actor and ERLANG language calculi which may be used or adapted for this work.

There is a series of works which build on each other that attempt to formalise ERLANG, namely those of Fredlund (2001), Claessen and Svensson (2005), Svensson and Fredlund (2007) and (Svensson, Fredlund & Earle, 2010). These formalisations occurred post-hoc, that is after the ERLANG language was created. Their aim was to create a faithful representation of the ERLANG compiler so as to study the behaviour of actors in local and distributed settings and encourage discussion on how the ERLANG language may be better understood and improved. Francalanza and Seychell (2013) model the basic ERLANG functions and also the tracing mechanism available in the ERLANG Virtual Machine (EVM). These previous works differ from ours as they do not discuss behavioural theories.

Similar to our goals, Thati (2003) and Gaspari and Zavattaro (1999, 2000) develop actor calculi to study equivalence theories. Thati develops an actor calculus, in this case to study the may testing preorder. However, while his aim is closest to ours, the actor language developed in this work is essentially a variant of asynchronous π-calculus with a type system enforcing discipline on the use of names. This means he forgoes actor features such as mailboxes and the *become* primitive is reduced to an input (without pattern matching). Gaspari and Zavattaro note that a lot of research, including equivalence theories, use process algebras, mathematical languages used for describing and verifying properties of concurrent systems such as CCS and π-calculus. However, they argue that such calculi are very different from the calculi developed for the actor model and real object-oriented and actor languages. They therefore develop an actor calculus and Labelled Transition Systems (LTS) that have a clean formal definition, like π-calculus, while capturing essential features specific to actors. They then use the language to study asynchronous bisimulation equivalence.

3.2 Behavioural equivalences

Given a language formalisation for ERLANG, we may then study *behavioural equivalences*. Behavioural equivalences are used to determine in which cases two subprograms offer similar interaction capabilities when run in the context of any larger program, and thus seem like an ideal candidate to be applied to refactoring tools. However, there is a large number of relevant properties which may need to be considered. This has led to the development of many different equivalences theories. An extensive description and a lattice of known behavioural equivalences and preorders over LTS, ordered by inclusion, can be found in the literature e.g. (van Glabbeek, 2001). To determine which behavioural theory should be used, we must first have a well-defined notion of what a safe or acceptable refactoring is. One definition, as defined in the original thesis about refactoring by Opdyke (Opdyke, 1992), is that behaviour is preserved if for the same sequence of inputs one receives the same sequence of outputs. However, this is not always sufficient, for example:

- Real-time software should preserve all temporal constraints, and
- Embedded software has memory constraints and power consumption which may need to be preserved by refactoring (Mens & Tourwé, 2004).

In our case we have decided to focus solely on the definition of behaviour preservation provided by Opdyke (Opdyke, 1992) mentioned previously. This definition is similar to that for the testing behavioural equivalences by De Nicola and Hennessy (De Nicola & Hennessy, 1984). In their work De Nicola and Hennessy state that if given two programs, A and B, behaviour is preserved if when A passes a set of tests, then B will also pass these tests. When applied to refactoring A would be the program before refactoring and B the program after refactoring. By generalising the set of tests to all possible tests, this condition may be reformulated as:

$$A \precsim_{test} B \text{ iff for all } T.\ \big(A \text{ passes } T\big) \text{ implies } \big(B \text{ passes } T\big)$$

In this way, we will be able to determine that the output after refactoring has not changed and thus the refactoring is correct.

Unfortunately this reasoning is impractical, as one cannot run the set of all possible tests in a finite amount of time. We therefore plan to investigate an alternative theory that facilitates reasoning about our testing equivalence. The technical development does not follow directly from previous work (Castellani & Hennessy, 1998; Boreale, De Nicola & Pugliese, 2002; Thati, 2003) as certain characteristics pertaining to actor systems complicate the development of our alternative preorder.

4 Conclusion

In this paper, we discussed the problem of ensuring behaviour preservation of a program after refactoring. While a number of approximation techniques exist, these cannot ensure that bugs are not introduced when refactoring. Thus, we focus on formal techniques, mainly behavioural equivalence, as a means of formally proving and thus ensuring that program behaviour was not modified when refactoring.

Using the literature surveyed as a starting point, in our work we develop a formal calculus for the Erlang language and a behavioural equivalence technique using the calculus which is suitable for refactoring tools. One possible starting point is using testing equivalences by De Nicola and Hennessy (De Nicola & Hennessy, 1984) as it is very similar to the definition of behaviour preservation found in the original thesis about refactoring by Opdyke (Opdyke, 1992). As reasoning using testing equivalence is often cumbersome, we are also tasked with developing an alternative theory for reasoning about testing equivalence, tailored for our Erlang calculus.

Acknowledgement

This study was carried out following the award of a STEPS scholarship which is part-financed by the European Union — European Social Fund (ESF) under Operational Programme II — Cohesion Policy 2007-2013, “Empowering People for More Jobs and a Better Quality of Life”.

References

Armstrong, J. (2007). *Programming Erlang - Software for a Concurrent World.* The Pragmatic Bookshelf.

Armstrong, J. (2010). Erlang. *Commun. ACM*, *53*(9), 68–75.

Boreale, M., De Nicola, R. & Pugliese, R. (2002). Trace and Testing Equivalence on Asynchronous Processes. *Inf. Comput.* *172*(2), 139–164.

Brown, C., Hammond, K., Danelutto, M. & Kilpatrick, P. (2012). A language-independent parallel refactoring framework. In *Proc. fifth work. refactoring tools* (pp. 54–58). WRT '12. New York, NY, USA: ACM.

Brown, C., Hammond, K., Danelutto, M., Kilpatrick, P., Schöner, H. & Breddin, T. (2013). Paraphrasing: Generating Parallel Programs Using Refactoring. In B. Beckert, F. Damiani, F. S. Boer & M. M. Bonsangue (Eds.), *Form. methods components objects* (Vol. 7542, pp. 237–256). Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Castellani, I. & Hennessy, M. (1998). Testing Theories for Asynchronous Languages. In *Fsttcs* (pp. 90–101).

Cesarini, F. & Thompson, S. (2009). *ERLANG Programming* (Media, Inc). O'Reilly.

Claessen, K. & Svensson, H. (2005). A semantics for distributed Erlang. *Proc. 2005 ACM SIGPLAN Work. Erlang - ERLANG '05*, 78.

De Nicola, R. & Hennessy, M. (1984). Testing equivalences for processes. *Theor. Comput. Sci.* *34*(1-2), 83–133.

Drienyovszky, D., Horpácsi, D. & Thompson, S. (2010). Quickchecking refactoring tools. In *Proc. 9th acm sigplan work. erlang* (pp. 75–80). Erlang '10. New York, NY, USA: ACM.

Francalanza, A. & Seychell, A. (2013). Synthesising Correct Concurrent Runtime Monitors - (Extended Abstract). In *Rv* (pp. 112–129).

Fredlund, L.-Å. (2001). *A Framework for Reasoning about Erlang Code* (Doctoral dissertation, Royal Institute of Technology, Stockholm, Sweden).

Gaspari, M. & Zavattaro, G. (1999). An Algebra of Actors. In P. Ciancarini, A. Fantechi & R. Gorrieri (Eds.), *Form. methods open object-based distrib. syst.* (Vol. 10, pp. 3–18). IFIP - The International Federation for Information Processing. Springer US.

Gaspari, M. & Zavattaro, G. (2000). An Actor Algebra for Specifying Distributed Systems: the Hurried Philosophers Case Study. In *Concurr. object-oriented program. petri nets, lect. notes comput. sci.* (pp. 428–444). Springer-Verlag.

Haller, P. (2012). On the Integration of the Actor Model in Mainstream Technologies: The Scala Perspective. In *Proc. 2nd ed. program. syst. lang. appl. based actors, agents, decentralized control abstr.* (pp. 1–6). AGERE! New York, NY, USA: ACM.

Haller, P. & Sommers, F. (2011). *Actors in Scala - Concurrent programming for the multi-core era.* Artima Press, California.

Hughes, J. (2007). QuickCheck testing for fun and profit. In *Proc. 9th int. conf. pract. asp. declar. lang.* (pp. 1–32). PADL'07. Berlin, Heidelberg: Springer-Verlag.

Karmani, R. K., Shali, A. & Agha, G. (2009). Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proc. 7th int. conf. princ. pract. program. java* (pp. 11–20). PPPJ '09. New York, NY, USA: ACM.

Li, H. & Thompson, S. (2005). Formalisation of Haskell refactorings. *Trends Funct. Program.* 265–280.

Li, H. & Thompson, S. (2007). Testing Erlang Refactorings with QuickCheck. In *19th int. symp. implement. appl. funct. lang. ifl 2007, lncs.* Freiburg, Germany.

Li, H. & Thompson, S. (2008). Testing Erlang Refactorings with QuickCheck. In O. Chitil, Z. Horváth & V. Zsók (Eds.), *Implement. appl. funct. lang.* (Vol. 5083, pp. 19–36). Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Li, H., Thompson, S., Orosz, G. & Tóth, M. (2008). Refactoring with Wrangler, updated: data and process refactorings, and integration with eclipse. *Proc. 7th ACM SIGPLAN Work. ERLANG.*

Mens, T., Demeyer, S., Bois, B. D., Stenten, H. & Gorp, P. V. (2003). Refactoring: Current research and future trends. In *Proc. third work. lang. descr. tools appl.* (pp. 120–130). Electronic.

Mens, T. & Tourwé, T. (2004). A survey of software refactoring. *Softw. Eng. IEEE Trans. 30*(2), 126–139.

Murphy-Hill, E. (2007). Programmer-Friendly Refactoring Tools. Other.

Murphy-Hill, E., Parnin, C. & Black, A. P. (2009). How we refactor, and how we know it. In *Proc. 31st int. conf. softw. eng.* (pp. 287–297). ICSE '09. Washington, DC, USA: IEEE Computer Society.

Opdyke, W. F. (1992). *Refactoring Object-oriented Frameworks* (Doctoral dissertation, Champaign, IL, USA).

Sagonas, K. & Avgerinos, T. (2009). Automatic refactoring of Erlang programs. *Proc. 11th ACM SIGPLAN Conf. Princ. Pract. Declar. Program. - PPDP '09*, 13.

Soares, G., Gheyi, R., Serey, D. & Massoni, T. (2010). Making Program Refactoring Safer. *IEEE Softw. 27*(4), 52–57.

Sultana, N. & Thompson, S. (2008). Mechanical verification of refactorings. *Proc. 2008 ACM SIGPLAN Symp. Partial Eval. Semant. Progr. Manip. - PEPM '08*, 51.

Sutter, H. & Larus, J. (2005). Software and the Concurrency Revolution. *Queue*, *3*(7), 54–62.

Svensson, H. & Fredlund, L.-Å. (2007). A more accurate semantics for distributed erlang. *Proc. 2007 SIGPLAN Work. Erlang Work. - Erlang '07*, 43.

Svensson, H., Fredlund, L.-Å. & Earle, C. B. (2010). A unified semantics for future Erlang. In *Proc. 9th acm sigplan work. erlang* (pp. 23–32). ACM.

Thati, P. (2003). *A theory of testing for asynchronous concurrent systems* (Doctoral dissertation).

van Glabbeek, R. J. (2001). The Linear Time-Branching Time Spectrum I - The Semantics of Concrete, Sequential Processes. In *Handb. process algebr. chapter 1* (pp. 3–99). Elsevier.

Ward, M. & Bennett, K. H. (1995). Formal methods to aid the evolution of software. *Int. J. Softw. Eng. Knowl. Eng. 5*(1), 1–18.