

Designing a Concurrency Aware Refactoring Framework

Maria Gouseti*
Bol.com, Utrecht, The Netherlands
mgouseti@gmail.com

Jurgen Vinju
Centrum Wiskunde & Informatica, Amsterdam
Eindhoven University of Technology, Eindhoven,
The Netherlands
Jurgen.Vinju@cwi.nl

1. INTRODUCTION

This extended abstract summarizes ongoing work, based on the first author's masters thesis [1], on a refactoring framework for the Java programming language which is correct in the presence of concurrency (i.e. Java multi-threading). We refer to the thesis for a precise positioning with respect to related work and detailed motivating examples.

A refactoring tool is a code transformation tool that restructures an existing body of code, altering its internal structure without changing its external behaviour. In this paper we focus on refactorings with the intent of moving an existing code snippet to another location. To ensure that the transformed code is correct, regression tests are often used [2, 3]. However, as parallel execution becomes more and more popular with the increasing production of multi-core processors and the use of distributed computing architectures, it is important to make sure that refactorings are concurrency aware and their correctness can be proven. Concurrency bugs are difficult to find through testing due to the infeasible amount of test runs necessary.

Schäfer et al. [4] use 5 motivating examples to demonstrate how refactored code that is correct in sequential execution may introduce bugs in concurrent execution. The authors defined three refactoring categories, organised by the theory used to prove their correctness: (i) *memory trace preserving* refactorings, which have to preserve data accesses on shared memory (e.g. MOVE METHOD); (ii) *dependence edge preserving* refactorings, which have to preserve data and synchronization dependencies (e.g. INLINE LOCAL); and (iii) *introducing new shared state* refactorings, which are left as future work since neither of the previous theories can be applied (e.g. CONVERT LOCAL VARIABLE TO FIELD).

To fix these refactoring tools, to be able to reason about their correctness, and to be able to generalise to more refactorings, we introduce an intermediate language called Synchronised Data Flow Graph (SDFG) and a framework called Concurrency Aware Refactoring with Rascal (CARR). SDFG

*Work performed while at Centrum Wiskunde & Informatica

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE 2016, Austin TX, USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

captures all the data and synchronisation dependencies of Java memory accesses of a program. We propose the existence of a set of code moving refactoring tools which preserve the shape of this intermediate representation to guarantee correctness. The CARR framework directly uses the preservation of the SDFG as a condition to the refactoring.

We are now working on showing how CARR produces correct concurrency aware refactoring implementations and we suggest that it could support lightweight proofs of correctness for a large range of refactoring tools. In this abstract we detail our goals and requirements and the principle architecture of CARR and the SDFG intermediate language.

2. GOALS

The state-of-the-art in correct concurrency refactorings is that different theories are used to prove different smaller classes of refactoring tools. Our goal is to find a common, simple theory which encompasses at least the published categories [4] and generalises to more refactoring tools. This theory could then be used to underpin the implementations of many concurrency aware refactoring tools.

1. Our first goal is to reproduce the results of Schäfer et al. [4] in a different context and generalise to at least include a fix for CONVERT LOCAL VARIABLE TO FIELD.
2. Our second goal is to support a wide range of refactoring tools, generalizing the state-of-the-art. Our main method is to introduce a reusable intermediate format into a framework which is parameterized by details specific to each refactoring.
3. Our third goal is to evaluate the architecture by applying it to refactoring tools which were previously incorrect in the presence of concurrency and have not yet been fixed.
4. Our final goal is to prove the correctness of each refactoring tool developed using CARR, one-by-one but reusing the framework to be able to reuse specific lemmas and limit the amount of work.

3. SOLUTION ARCHITECTURE

Figure 1 depicts an overview of the CARR framework for constructing correct refactoring tools.

The basic conceptual idea is to convert the Java code under refactoring both before and after applying the proposed code transformation to an intermediate representation called the Synchronised Data Flow Graph (SDFG) language.

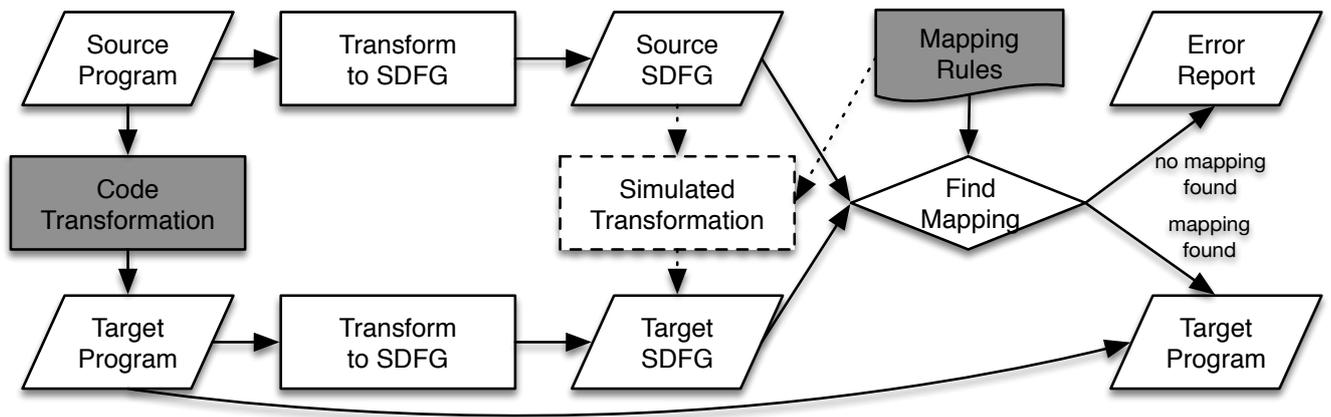


Figure 1: Concurrency Aware Refactoring Architecture —Flow Graph of CARR. The light parts are reusable infrastructure, the dark parts are specific per refactoring tool. Intermediate representations are considered I/O for the refactoring tool engineer, not for the end-user programmer.

The SDFG format captures all data and synchronisation dependencies. These two program representations can then be compared to one another to check for correctness. If the comparison finds the two representations equivalent with respect to the dependencies between memory accesses, then the refactoring may go through, otherwise not.

A fast and practical implementation of CARR would be to simulate the code transformation on the SDFG level directly, before actually applying the refactoring to code. This enables the same post-condition checking. However, to avoid possible confusion, for the current presentation we take a more conceptual point of view and we assume the compared SDFG representations are acquired from the source and target code respectively.

Portability of refactoring tools between programming languages may be easier using CARR; by providing a new front-end mapping to SDFG and reusing the mapping rules. However, the obligation to prove the mapping rules correct for the new language remains.

The linchpin design element in this architecture is “Find Mapping” which decides if the two SDFG graphs are semantically equivalent in terms of flow and synchronisation semantics. The possibilities for mapping semantically equivalent parts of the SDFGs are specified by a (small) set of equivalence inference rules. These rules are unique for each refactoring tool: they model the unique properties of each refactoring tool. On the one hand, the rules must be re-considered for every new refactoring tool. On the other hand, the rest of the CARR infrastructure may be reused as soon as the new mapping function has been proven correct and implemented. In this abstract we describe one such mapping function as an example.

The key enabler for CARR as a reusable infrastructure is a sound and complete mapping of programming language syntax and semantics to the intermediate representation. The code for the transformation from Java to SDFG is based on the Eclipse JDT compiler¹ and the Rascal M3 front-end [5].

4. SYNCHRONISED DATA FLOW GRAPH

¹<http://www.eclipse.org/jdt>

An engineering challenge in constructing refactoring tools for real programming languages is to capture the semantics for the complete language in a faithful manner. This motivates the mapping of the Java language to a simpler and more manageable intermediate format, clearly separating arbitrary details of syntax and (static) semantics, from the concepts of interest (data flow and synchronisation).

4.1 Definition

Figure 2 defines the intermediate language —SDFG— as inspired by OFG [6]. This language strips a programming language like Java from the structural features and maintains only a set of “declarations” and a set of “statements”. The first identify shared data locations (fields) and inter-procedural flow points (constructors and methods) while the latter reflect only the dependencies between memory accesses from the original Java code. Such dependencies include the data and synchronisation dependencies from the Java Memory Model (JMM) applied on local memory accesses in addition to the shared ones, as well as other dependencies that reflect memory allocation and the beginning and ending of methods. Each statement can also be read as a binary constraint between two code points or between a code point and a value². The `loc` data-type is used throughout SDFG to represent unique source code artefact locators [5].

4.2 Mapping Java to SDFG

To extract the implicit dependencies, the converter³ traverses the AST and passes in every node the current state which is updated and returned. The current state contains: (i) a set of the `Stmts` that were gathered in that specific path, (ii) a map with the identifiers of the last visible `assign(.)` or `change(.)` of a variable declaration, (iii) a second map that contains the identifiers of the last change of a class which corresponds to the last visible `change(.)` and (iv) a relation that contains a tuple with the lock declaration and the identifier of an acquire action.

The two types of AST nodes that the converter visits are the expressions and the statements. Both types return the

²Values are symbolic data values such as numbers or strings

³<https://github.com/gmarouli/SDFG>

```

data Program
= program(set[Decl] decls, set[Stmnt] stmts);

data Decl
= attribute(loc id, bool volatile)
| method(loc id, list[loc] formals, loc lock)
| constructor(loc id, list[loc] formals);

data Stmnt
= read(loc id, loc variable, loc depId)
| assign(loc id, loc variable, loc depId)
| change(loc id, loc typeDecl, loc dataDepId)
| acquireLock(loc id, loc lock, loc depId)
| releaseLock(loc id, loc lock, loc depId)
| create(loc id, loc constructor, loc actParId)
| call(loc id, loc rec, loc method, loc actParId)
| entryPoint(loc id, loc method)
| exitPoint(loc id, loc method);

```

Figure 2: Definition of the Synchronised Data Flow Graph language (SDFG) in Rascal [7].

exception state, which maps an exception with the current state and keeps it until a **catch** statement is found. In case of an expression, the converter returns a set of potential **Stmnts** which refer a potential read of the current variable. Finally, in case that a statement is **continue**, **break** or **return** the current state is saved and returned to its parent node until it is time to be used. For instance, a state stored after a **break** statement is stored until exiting a loop or a **case** from a **switch** statement.

Preprocessing steps can make the dependency extraction more accurate, for example the map that contains the last change of an object will be different if pointer analysis and detection of methods that do not mutate state proceed it. This shows the strength of this tool since it's accuracy can be improved by using better algorithms to resolve dependencies.

4.3 SDFG equivalence

Given that a dependency is a transitive relation between two **Stmnts**, we say that two SDFG programs simulate each others dependence relation if they preserve the indirect dependencies and we can argue that the changed dependencies do not change behaviour.

Inference rules are used to model the expected changes and map the refactored program to an inferred SDFG program. Then, the original SDFG program is compared with the inferred one. If the inferred program is a subset of the original one, we guarantee that no unexpected dependencies are introduced and, if the inference rules are correct, no dependencies are lost. Consequently, proving the correctness of the refactoring lies on proving the correctness of the inference rules that model the anticipated changes.

5. CARR

The general algorithm used in the CARR implementation consists of the following steps that were illustrated in Figure 1. The first and the last step are refactoring specific.

Code transformation, generating new identifiers when needed and keep a map with the correspondence between the original and the new identifiers.

Transform to SDFG both programs: the original and the refactored one.

Find mapping by applying the mapping rules on the

transformed programs and either reject or accept the refactoring.

6. EXAMPLE - MOVE METHOD

The MOVE METHOD refactoring first applies a transformation to the method code and then applies another transformation to the rest of the code and the updated method code. There are three ways that a method can be accessed after it has been moved to another class: (i) the method is static, (ii) the method has as a parameter the destination class so the receiver and the parameter can be swapped, and (iii) the destination class is a field of the source class of the method. Consequently, the rules concerning the refactored code of the method are:

- There should be no **assign**(\cdot) that refers to **this** or to the field of the class used as receiver in case of parameter swap and field access respectively.
- The fields should have read dependencies to the new parameter or the qualified name.
- In case that the destination class was a parameter, the dependencies to the parameter should be replaced by dependencies to **this**.
- The receiver of a method call should not be the value **null**.

CARR requires that every method call to this method is updated to match its new declaration. The previous rules are formalised in the following inference rules. The inference rules model how the **Stmnts** from the refactored SDFG program can be used to generate the original program.

In the following rules md is the original method declaration and md' is the refactored one. The next rules apply to all the cases. The inference rules in which the premise and conclusion are the same are omitted. RP and IOP stand for Refactored Program and Inferred Original Program respectively.

$$\frac{RP \models \text{entryPoint}(id, md')}{IOP \models \text{entryPoint}(id, md)}, \frac{RP \models \text{exitPoint}(id, md')}{IOP \models \text{exitPoint}(id, md)} \quad (1)$$

In the case of static methods the inference rules for the method code and the whole program are 2 and 3 respectively.

$$\frac{RP \models \text{read}(id, \text{destinationClass}, \text{dep})}{IOP \models \text{read}(id, \text{sourceClass}, \text{dep})} \quad (2)$$

$$\frac{RP \models \text{call}(id, \text{destinationClass}, md', \text{arg})}{IOP \models \text{call}(id, \text{sourceClass}, md, \text{arg})} \quad (3)$$

In the case of the parameter swap, the following inference rules refer to the method code, where p' refers to the new parameter of the *sourceClass* and p to the original parameter. The command **abort** represents the immediate rejection of the refactoring transformation due to an invalid **Stmnt** with respect to Java semantics.

$$\frac{RP \models \text{read}(id, \text{this}, \text{dep})}{IOP \models \text{read}(id, p, \text{dep})}, \frac{RP \models \text{read}(id, p', \text{dep})}{IOP \models \text{read}(id, \text{this}, \text{dep})} \quad (4)$$

$$\frac{RP \models \text{assign}(id, \text{this}, \text{dep})}{IOP \models \text{abort}} \quad (5)$$

$$\frac{\text{RP} \models \text{acquireLock}(\text{id}, \mathbf{this}, \text{dep})}{\text{IOP} \models \text{acquireLock}(\text{id}, \text{p}, \text{dep})} \quad (6)$$

$$\frac{\text{RP} \models \text{releaseLock}(\text{id}, \mathbf{this}, \text{dep})}{\text{IOP} \models \text{releaseLock}(\text{id}, \text{p}, \text{dep})} \quad (7)$$

$$\frac{\text{RP} \models \text{acquireLock}(\text{id}, \text{p}', \text{dep})}{\text{IOP} \models \text{acquireLock}(\text{id}, \mathbf{this}, \text{dep})} \quad (8)$$

$$\frac{\text{RP} \models \text{releaseLock}(\text{id}, \text{p}', \text{dep})}{\text{IOP} \models \text{releaseLock}(\text{id}, \mathbf{this}, \text{dep})} \quad (9)$$

Inference rules for the whole SDFG program, where *rec* is the original receiver of the method, *arg* is the parameter being swapped and *a* all the other arguments of the method call:

$$\frac{\text{RP} \models \text{call}(\text{id}, \text{arg}, \text{md}', \text{rec}) \quad \text{RP} \models \text{call}(\text{id}, \text{arg}, \text{md}', \text{a})}{\text{IOP} \models \text{call}(\text{id}, \text{rec}, \text{md}, \text{arg}) \quad \text{IOP} \models \text{call}(\text{id}, \text{rec}, \text{md}, \text{a})} \quad (10)$$

In case of fields access, the inference rules for the method code, where *p'* refers to the new parameter of the *sourceClass* and *f* refers to the field used as the receiver of the class:

$$\frac{\text{RP} \models \text{read}(\text{id}, \text{p}', \text{dep})}{\text{IOP} \models \text{read}(\text{id}, \mathbf{this}, \text{dep})} \quad (11)$$

$$\frac{\text{RP} \models \text{read}(\text{p}', \text{p}', -), \text{assign}(\text{f}_a, \text{f}, \text{p}')}{\text{IOP} \models \mathbf{abort}} \quad (12)$$

Inference rules for the whole SDFG program:

$$\frac{\text{RP} \models \text{read}(\text{f}_r, \text{f}, \text{rec}), \text{call}(\text{id}, \text{f}_r, \text{md}', \text{a}), \text{call}(\text{id}, \text{f}_r, \text{md}', \text{rec})}{\text{IOP} \models \text{call}(\text{id}, \text{rec}, \text{md}, \text{a})} \quad (13)$$

In summary, the above example inference rules are all that is needed to instantiate CARR for MOVE METHOD. The size of the Rascal code implementing these rules is comparable to the size of the above definition in L^AT_EX source code. These rules introduce a proof obligation of soundness (each rule guarantees semantic preservation) and completeness (the rules together enable a match in all relevant circumstances). Incomplete rule sets will lead to spurious rejections (no match can be made and the refactoring is rejected), while an unsound rule will allow the refactoring to introduce bugs. So soundness should be the focus of theoretical investigation, while completeness is a matter of experimental evaluation.

7. FUTURE WORK

Proving correctness of each refactoring tool is based on the use of Separation Logic and basic Hoare Logic. Proof sketches can be found in the first author's masters thesis [1].

Updating objects leads to inaccuracy. Currently if a method is called on an object or a field is changed, we assume that every object of that type has changed too. This results in rejecting refactorings that do not change behaviour. This may be solved in the future by applying two extra steps before transforming to SDFG. The first step would be analysing aliases and the second to detect if a method changes the state outside of its scope.

Intra-procedural analysis limits scope of possible refactoring tools. If added, even more different kinds of refactorings could be considered.

Full integration into the Eclipse refactoring user-interface is left as future work. For now we can trigger CARR programmatically, which allows to evaluate its contributions adequately.

8. CONCLUSION

In this work we focus on understanding when the behaviour of refactored code is changed in the presence of concurrency, and provide a configurable formal model in the format of an intermediate language, SDFG, that can be used for semantic verification and implementation of code movement refactorings.

We implemented the converter from Java to SDFG and integrated it to a refactoring tool called CARR to make the tool concurrency aware. Our next goal is to evaluate the architecture experimentally.

9. REFERENCES

- [1] M. Gouseti, "A general framework for concurrency aware refactorings," Master's thesis, Universiteit van Amsterdam, 2014.
- [2] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 185–194. [Online]. Available: <http://doi.acm.org/10.1145/1287624.1287651>
- [3] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 147–162, 2013. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/TSE.2012.19>
- [4] M. Schäfer, J. Dolby, M. Sridharan, E. Torlak, and F. Tip, "Correct Refactoring of Concurrent Java Code," in *ECOOP 2010 — Object-Oriented Programming*, ser. Lecture Notes in Computer Science, T. D'Hondt, Ed. Springer Berlin Heidelberg, 2010, vol. 6183, pp. 225–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14107-2_11
- [5] A. Izmaylova, P. Klint, A. Shahi, and J. J. Vinju, "M3: an open model for measuring code artifacts," *CoRR*, vol. abs/1312.1188, 2013. [Online]. Available: <http://arxiv.org/abs/1312.1188>
- [6] P. Tonella and A. Potrich, "The Object Flow Graph," in *Reverse Engineering of Object Oriented Code*, ser. Monographs in Computer Science. Springer New York, 2005, pp. 21–41. [Online]. Available: http://dx.doi.org/10.1007/0-387-23803-4_2
- [7] P. Klint, T. van der Storm, and J. Vinju, "Easy meta-programming with rascal," in *Generative and Transformational Techniques in Software Engineering III*, ser. Lecture Notes in Computer Science, J. a. Fernandes, R. Lämmel, J. Visser, and J. a. Saraiva, Eds. Springer Berlin Heidelberg, 2011, vol. 6491, pp. 222–289. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18023-1_6