A Formal Framework Supporting Unrestricted Software Changes in Object-Oriented Concurrent Systems*

Olaf Owe, Jia-Chun Lin, and Ingrid Chieh Yu

Department of Informatics, University of Oslo
E-mails: {olaf,kellylin,ingridcy}@ifi.uio.no

Abstract

Program evolution may reveal bad design decisions, misunderstandings, or erroneous code and specifications. Problems made early may not be discovered until much later. Non-trivial changes of old code may be needed, and flexibility in making changes is essential. We propose a framework for reasoning about unrestricted program/specification changes, focusing on the challenges of concurrent and object-oriented programs.

Motivation

Program development is in general a complicated process where many kinds of mistakes can be made over time. There can be bad design decisions, unclear specifications, misunderstandings, or erroneous code and specifications. Problems made early may not be discovered until much later. Redoing code made at an early stage in the software development may affect many parts of the overall system. Thus making changes in order to correct problematic decisions may create new problems that are hard to foresee. These kinds of problems are severe in the setting of concurrent programs when the interaction of the different concurrent units is complicated, and also in the setting of object-oriented programs, due to inheritance, dynamic binding and code reuse.

A systematic approach in which the consequences of a software change can be formalized, would be advantageous. Formal methods could be helpful in supporting specification and analysis of program properties. However, formal methods are mainly oriented towards developing correct specifications and programs rather than the process of redoing earlier decisions. It is therefore interesting to look at formal frameworks with support for unrestricted software changes, and such that the framework can detect possible consequences.

A trivial approach to reasoning about program changes is to re-verify and reprove all results whenever a change has been made. However this is time consuming and it is an expensive solution, especially for large software systems. Ideally we would like to reprove as little as possible, without losing soundness. And we would like to consider the setting of concurrent and object-oriented programs, which is both relevant in the software industry and challenging. The simplicity of a reasoning framework for software changes depends on the choice of specification and reasoning mechanisms as well as the language constructs and their semantics. For some programming paradigms, like shared variable concurrency, it is hard to analyze the effect of software changes, even with an advanced reasoning framework. We will therefore consider asynchronously communicating concurrent objects, since this setting offers compositional verification, and we consider program assertions over the communication history, since this captures all interactions and offers a measure of possible side-effects.

*This work was done in the context of the EU projects H2020-644298 *HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems*, FP7-610582 *Envisage: Engineering Virtualized Services*, and FP7-ICT-2013-X *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations*.

Related Work

In formal methods the notion of refinement is used to reflect software development. A refinement is in general leading from a design with certain properties to a design which preserves these properties, while adding more detail. In this way refinement is semantics-preserving. Certain refinement logics support the introduction of (additional) error values, thereby semantics is preserved as long as no errors appear. Banach et al have argued for the need of refinement-like steps that go beyond the limitation of semantics-preserving development [1]. However, their approach does not support analysis of program properties.

In the setting of object-oriented programs with inheritance, behavioral subtyping is the most common reasoning approach, restricting subclasses to obey the super-class specifications [7]. This means that subclasses must preserve behavior in some sense. Lazy behavioral subtyping [2] relaxes this condition; only behavior that is needed to verify local calls in a superclass must be respected by a subclass redefining the method. This gives added flexibility, allowing a larger class of changes without breaking the requirements. A number of works on asynchronously communicating concurrent objects, partly by the authors of this paper, consider certain forms of software and/or specification changes: The concept of dynamic software updates allows changes to superclasses [6]. Interface abstraction allows reasoning about remote calls to rely on the declared interface of the callee. This means that changes in a (super)class implementation may be done as long as the stated interface support is respected, and as long as subclass reasoning is not affected.

A calculus allowing changes to methods, (super)classes and interfaces is presented in [4]. The calculus can be seen as a generalization of lazy behavioral subtyping. Program properties, represented by Hoare triples, are classified in two categories for each class C, representing the verified ones and the unresolved (unverified) ones, $\mathcal{U}(C)$. The set of verified properties of a given class C and method m is denoted $\mathcal{G}(C, m)$. When the set of unverified program properties is verified (i.e., $\mathcal{U}(C)$ is empty) the class is found to be correct in the sense that all pre/post method specifications are satisfied by the corresponding implementation in a class as well as those in interfaces supported by the class. Changes in code or specifications may affect both categories. However, a program requirement added to $\mathcal{U}(C)$ may be impossible to verify (in case the Hoare triple is not satisfied), and it will then remain in $\mathcal{U}(C)$, and there is no guarantee that this problem is detected.

The approach in [3] addresses transformation of classes and allow classes in the middle of a class hierarchy to be changed. Modifications are archived by means of update operations *modify* and *simplify*. The modify operations extend class definitions, allowing code such as new fields, method definitions, guarantees, and interfaces to be added to classes, and existing methods to be redefined. The simplify operations allow redundant methods to be removed from class definitions. The approach does not classify classes using $\mathcal{G}$ and $\mathcal{U}$ such as in [4], rather, for each update applied to a class, all verification work is done to methods affected by the update. However, the approach is limited to interface additions, i.e., implemented interfaces can not be removed or modified and methods can only be changed if the guarantees are strong enough to satisfy the constraints of the implemented interfaces. And superclass requirements needed to handle local calls are imposed on subclasses, as in [4].

Another line of works consider proof reuse, including partial reuse of proofs of earlier verified properties. This may require some storage of proof outlines or non-trivial verification steps. This means that when a (part of a) program is corrected, one may try to rerun previous proofs to alleviate the verification burden [9]. The notion of abstract method calls allows reuse of abstract proof outlines, for a fixed method body, while their instances may need further work when other methods or requirements are changed [5] These approaches simplify the verification

task of evolving programs. The amount of proof reuse can be balanced against the amount of automation. Efficiently automated proof need not be reused while interactive proofs could benefit from reuse, if possible. In our approach we will be oriented towards a language with a high degree of automation of verification conditions and proof reuse is therefore not in our focus. For simplicity we will therefore not consider techniques for proof reuse.

Approach

In this paper we try to find a good framework for reasoning about unrestricted program changes, focusing on the challenge of concurrent and object-oriented programs. We use interface abstraction since this limits the visible effect of low-level changes. Furthermore we allow multiple invariants for each class/interface since this makes it meaningful to add new invariants for old classes upon need. In program evolution, specifications should be allowed to evolve. Finally we consider primitives for changing old or new interfaces or classes, including methods, invariants, inheritance, and interface support. In general a software modification will consist of a sequence of several primitive changes constituting a meaningful modification. We assume type correctness, and therefore fields may only be removed when no longer in use. Thus one must modify all methods and invariants using a field before removing it.

Our primitives allow unrestricted changes of code (assuming type correctness). This means that one may write combinations of code and invariants that are inconsistent, for instance when a class does not satisfy the requirements of its interface(s). The framework will help in detecting such inconsistencies so that they may be resolved. In order to determine the consequences of changes in a (super)class the framework needs to keep track of dependencies of local calls. In particular for reasoning about inheritance, we build on the approach of *behavioral interface subtyping* [8] where each class is only required to satisfy its invariant(s) and interface specifications and any other local specifications given in the class. This means that a method redefined in a subclass may break the requirements of the superclass, even the minimal requirements imposed in the case of lazy behavioral subtyping. This opens up for more liberal modifications than earlier work based on lazy behavioral subtyping as no superclass requirements are imposed on a subclass. The consistency of a class is determined by looking at the class itself, its interface(s), and any reused code from superclasses. For a software modification one must first determine the affected code, and for each class containing such code one must re-verify the affected parts.

References

[1] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Science of Computer Programming*, 67(2âĂŞ3):301 – 329, 2007.

[2] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[3] J. Dovland, E. B. Johnsen, O. Owe, and I. C. Yu. A proof system for adaptable class hierarchies. *Journal of Logical and Algebraic Methods in Programming*, 84(1):37 – 53, 2015.

[4] J. Dovland, E. B. Johnsen, and I. C. Yu. Tracking behavioral constraints during object-oriented software evolution. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2012.

[5] R. Hähnle, I. Schaefer, and R. Bubel. Reuse in software verification by abstract method calls. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2013.

[6] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of Lecture Notes in Computer Science, pages 15–30. Springer, June 2005.

[7] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACMTransactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[8] O. Owe. Verifiable programming of object-oriented and distributed systems. In L. Petre and E. Sekerinski, editors, *From Action System to Distributed Systems: The Refinement Approach*. CRC Press, 2015. To appear.

[9] W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 284–293. Springer-Verlag, 1993.