

Concern-specific annotation languages to support static detection of bugs in Java-like programs

Lieven Desmet¹, Bart Jacobs^{1*}, Frank Piessens¹, Wolfram Schulte², Jan Smans^{1*}, Dries Vanoverberghe¹

¹DistriNet, Dept. Computer Science, K.U.Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{lieven,bartj,frank,jans,driesv}@cs.kuleuven.be

²Microsoft Research
One Microsoft Way, Redmond, WA, USA
schulte@microsoft.com

Formal methods have long been recognized as an important tool to help ensure quality of software. Among the most successful and widespread formal methods are *type systems*. Type systems can be seen as lightweight specification formalisms with a decidable notion of validity (implemented by the type checker), yet with sufficient specification power to guarantee the absence of certain classes of bugs. For Java-like languages, the research community has designed type systems to deal with a variety of bugs, including concurrency related bugs, null pointer dereferences, security related bugs, encapsulation related bugs, and so forth.

But as more and more of these *concern-specific* type systems are proposed, it becomes more important to be able to combine them, or even empower developers to pick the combination of type systems they want to use for a given development effort. Only recently, the idea of optional and pluggable type systems has been put forward in the context of Java-like languages. Also, to ensure decidability, type systems rule out certain correct programs.

Over the past years, we have explored the use of more powerful kinds of program annotations for Java and C# to deal with the same classes of bugs that type systems typically address. Our annotations have no run-time effects: they can be added as stylized comments. Hence, it is easy to combine annotations for different concerns, and annotations are always optional. An annotated program is translated to first-order verification conditions. If the verification conditions can be proven, then a certain class of bugs is guaranteed not to be present in the original annotated program. Our approach supports a trade-off between annotation overhead and precision of the verification: by providing more annotations the programmer can make more (correct) programs pass the verifier. This is in sharp contrast with type systems, where the precision of the verification is fixed with the design of the type system.

* Bart Jacobs and Jan Smans are Research Assistants of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen)

We have instantiated this idea in three different settings:

- to deal with concurrency related bugs (data races and deadlocks), as described in detail in the papers [3–5].
- to deal with stack-inspection based sandboxing related bugs, as described in detail in the papers [6, 7].
- to deal with data dependency related bugs as described in detail in the papers [1, 2].

The common theme in these systems is that we design a concern-specific annotation language (different for each of the three systems), and translate annotated programs to a general purpose specification formalism, that is then fed to an automatic prover.

For each of these systems, we have built a prototype verifier, and have gained some experience in verifying medium sized software systems. Our case studies include a chat server, and an open-source web mail application. Experience shows that, while verification of the annotations is undecidable in general, realistic programs can be verified in a reasonable amount of time and with a reasonable annotation overhead.

In this tutorial paper, we report on our experiences in designing these three systems, and highlight the commonalities in their design. The design of a concern-specific annotation language is done in six steps:

1. Pick the concern to be addressed, e.g. data race freedom.
2. Design a set of conceptual rules that address the concern, e.g. a thread t can only access a field of object o if o is either thread local or t has locked o .
3. Decide on the conceptual model and its properties, e.g. accessibility is modeled using access sets (sets of objects accessible to a thread) with the property that access sets will never intersect in any run of a correct program.
4. Decide on the ghost state tracked by the verification and on the necessary annotations, taking into account modularity requirements.
5. Define the translation from annotated programs to verification conditions.
6. Prove soundness.

From our experience in these three systems, we expect the same approach to be useful for other concerns, and for future work we plan to apply the same approach for dealing with the concern of limiting resource consumption on mobile devices.

References

1. L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, Static Verification of Indirect Data Sharing in Loosely-coupled Component Systems. In *SC 2006*, (W. Löwe and M. Südholt, eds.), vol. 4089/2006, Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2006, pp.34-49.
2. L. Desmet, F. Piessens, W. Joosen, and P. Verbaeten, Bridging the Gap Between Web Application Firewalls and Web Applications, to appear in the proceedings of Formal Methods in Security Engineering 2006.

3. B. Jacobs, K. Leino, and W. Schulte, Verification of multithreaded object-oriented programs with invariants, SAVCBS 2004 specification and verification of component-based systems: workshop proceedings (Barnett, M. and Edwards, S.H. and Gianakopoulou, D. and Leavens, G.T. and Sharygina, N., eds.), vol 04-09, Technical report, Department of Computer Science, Iowa State University, pp. 2-9, 2004
4. B. Jacobs, K. Leino, F. Piessens, and W. Schulte, Safe concurrency for aggregate objects with invariants, Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods (Aichernig, B.K. and Beckert, B., eds.), pp. 137-146, 2005
5. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs. Accepted at the Eighth International Conference on Formal Engineering Methods (ICFEM 2006), Macau, October 29-November 3, 2006.
6. J. Smans, B. Jacobs, and F. Piessens, Static verification of code access security policy compliance of .NET applications, Proceedings of the 3rd International Conference on .NET (Skala, V. and Nienaltowski, P., eds.), pp. 1-12, 2005
7. J. Smans, B. Jacobs, and F. Piessens, Static Verification of Code Access Security Policy Compliance of .NET Applications, Journal of Object Technology 5 (3), pp. 35-58, April, 2006.