Software Verification

Peter H. Schmitt Mattias Ulbrich

Karlsruhe Institute of Technology (KIT)
October 29, 2010

1 Our view of the field of Verification

When a technology reaches a higher level of maturity, its development begins to diversify and to differentiate. In 1885 engineers were happy to build the first car powered by a gasoline engine. Today we see a whole spectrum of vehicles ranging from city cars, limousines, off-road vehicles, sports cars, buses, trucks, etc., that share some basic technologies but greatly differ in details. Software Verification has entered this phase of diversification and we need to start this exposee with a rough classification. On the top level we distinguish between model verification and program verification.

By **model verification** we understand the formal analysis and modelling of systems, algorithm or software at a high level of abstraction using mathematical notions like sets, sequences, relations. Representatives of this category are for instance the B-method, the Z approach, or abstract state machines (ASM). With these, often some concept of refinement is involved, that formalizes the transition from an abstract model to a more concrete model. The goal of these methodologies may be characterized by the label: correctness by construction. Another set of methods use temporal logic model checking (e.g., Promela, CSP). Methods and tools in this category support system development but do not look into the actual implementing code. They are good for eliminating design flaws at a very early stage.

Biased by our own interests and competence we will say no more on this kind of verification and concentrate on **program verification**. This approach presupposes the existence of executable program code together with a specification. Usually the specifications come as annotations in the code written in a specification language closely related to the programming language in use.

Before we can comment on the usability of program verification, we need to have an idea to what use it shall be put. Here is a first coarse classification of the use cases of formal methods in programming:

1. **Find bugs:** Formal methods are used to analyse a program with no or little additional annotations to identify possible runtime anomalies.

2. **Increase confidence:** Code is proved (or at least formally checked) to satisfy the specification given through its annotations. The annotations need not completely cover a full functional specification, but may address specific issues, such as the absence of runtime exceptions, valid sequence of method invocations on objects, or ensuring that memory is correctly initialised. This is sometimes referred to as lightweight verification.

3. **Prove correctness:** To establish the highest level of confidence, full functional verification can be performed. The annotation overhead is significantly higher both in length and in complexity of the specifications. The costs to achieve ultra-dependable software are still very, very high.

In all three areas, formal verification has successfully passed the *proof-of-concept* stage. On the other hand, it has to be admitted that no clear business cases have so far emerged in any of these categories.

Some ten years ago, the verification community took the challenge to verify not only examples in *toy languages* but to face the complexity of existing *real programming languages*, especially Java, Ada, C# and C.

There have been many projects in the field of Java and Spec# verification. It seems that modelling the sequential semantics of modern, well-typed languages with automatic memory management into logic has been solved satisfactorily. There are no principal obstacles caused by the complexity of these languages to full functional verification.

Verification of programs written in C has started with some delay. The semantics of C is less well-defined. This makes the formalisation and subsequent verification of C programs significantly more complex than that of Java programs even if thorny issues like pointer arithmetics are excluded. On the other hand, programs written in C are much more error-prone, which makes program verification even more worthwhile.

The VCC project and L4.verified have proved that C formalisation up to a rather detailed degree is possible. HAVOC and SLAM and CBMC/LLBMC have proved that lightweight methods for C are able to detect plenty of bugs.

Another positive development is the dramatic improvement in the power of the available decision procedures to discharge the proof obligations arising from program verification. The wide-spread adoption of the SMT interface has boosted this development by increasing competition.

However, there are topics that still await a satisfactory solution.

1. *The problem of framing:* Which memory locations are affected by a certain piece of code? This is badly needed for good modularisation.

2. *The problem of concurrency:* How do several threads which run in parallel affect the shared data? How can one guarantee correct programs if lock and channel communication is used?

3. *Application of data types:* The link between model world and implementation is often still underdeveloped. But data abstraction most often asks

for mathematical structures like trees, sets, sequences, ... To work with them and connect them with actual implemented data structure is under investigation.

These issues are not connected to a particular language but address cross-cutting research problems in their own right.

In our opinion research on **issue 1** has made greater progress than the others. Several promising candidate solutions for dealing with the frame problem have been put forward including data groups, dynamic frames, separation logic, ownership, and regional logic.

We believe that verification of concurrent programs, **issue 2**, is still not entirely understood. The treatment of cooperating threads that communicate in a well-defined and specified manner (using channels, locks or assume-guarantee contracts) is being worked on, but still lacks efficiency. Chalice has been developed as a verification language to model concurrency. Is this a symptom that existing programming languages do not offer the *right* means for concurrent programming?

From a wider perspective **issue 3** might be seen as a bridge over the gap between program verification and what we called model verification in Section 1. Taking the verification of the Schorr-Waite algorithm as an example, there exist about a dozen papers solving this task using a number of different tools, it certainly makes sense to separate the verification of the correctness of the algorithm from the verification of the implementation. At the moment none of the proposed solution accomplishes this completely.

2 What Might Happen

Turning to the question of *usability* obviously the additional effort that is needed to apply formal verification will be a major criterion for adopting it or not. Let us again look at the three possible goals of formal verification from the first enumeration in Section 1 now augmented with comments on the overhead needed for their application.

Usage	**Annotation overhead**	**Technologies**
Extended debugging	no/little additional annotations	Abstract interpretation, Software model checking, Type systems
Increase confidence	partial functional specification ($< 100\%$)	Runtime assertion checking, Extended static checking
Ultra dependability	full functional specification ($\gg 100\%$)	Theorem provers, Interaction

As far as debugging technologies are concerned, we understand that static analyses, dynamic (i.e., run-time) formal methods and similar techniques are pretty evolved and are about to be deployed to mass markets. The inclusion of code contracts in Visual Studio is good evidence for that.

One would have expected that full functional verification would be enthusiastically welcomed in safety critical areas as, for instance, avionics. However, progress is very slow. Formal method interest groups have formed themselves within the relevant bodies, but have not yet gained major influence. It is hard to tell if, when, in what form verification of ultra-dependable systems will be applied in the real world. From a scientific perspective, however, it seems the right thing to do. You never know for sure. Maybe the cybercrime debate will change things?

The middle segment in our coarse classification is set apart from the first category by an increased effort of writing contracts. We can only speculate how much overhead will be acceptable to how many people. We are, however, not too pessimistic.

Is the goal of the future to facilitate the use of simple, less powerful lightweight methods or advances in more powerful heavyweight approaches which require more effort?

Perhaps, it is a goal to move the "border" between light and heavyweight such that properties which require considerable effort today can be automatically verified in the future without much doing on the part of the verifying person.

Another development that so far has not gained momentum may be described by the headline *designed for verification*, i.e., the design of systems and programs in such a way that they are easy to verify.

3 Next Steps

In this section we propose a short- and medium-range agenda for the next steps to be pursued in the development of usable program verification systems.

1. *Specification Languages*
   There certainly are still research issues in the area of framing, invariant semantics, and data abstraction, but we know enough already to implement solutions into existing specification languages. We particularly think of JML in this context.

   To gain a broader acceptance, specification languages should cover a wide range of applications. It should be possible to use the same specification mechanisms (or, even better, the same specifications) for runtime checking, static analyses, test case generation, heavyweight approaches, and so on. Although this is advertised already today, interpretations of the semantics of one language can differ considerably between approaches.

2. *Verified Libraries*
   It is essential for a wide-spread use of program verification that library functions can be used without restraints and preferably without additional overhead. There may still be one or the other research question involved in this task, but on the whole it is just a tremendous routine job. This is

not a task for an academic research project. Also the community efforts, e.g., by the JML community, so far did not make much headway.

3. *Domain Specific Applications*
   It is the rationale behind this item that in more specific domains, specifications can be directly formulated in the vocabulary used in this domain and thus be accessible to an audience not trained in formal methods.

   For full functional verification, the specification is often longer than the actual implementation, diminishing the documentation character of the specification. Domain specific descriptions are usually conciser and more intuitive to read. Such specifications can be shorter and less complex and, at the same time, as precise as a general specification.

4. *Bridge the Gap between Model and Program Verification*
   The use of abstract data types for *model* and *ghost* variables, fields and methods in annotation languages is a first step to move from code to a more abstract level. More needs to be done along these lines.

   Formal modelling methodologies used in early design stages should seamlessly pass into program verification. This would possibly be done using a notion of refinement. It is already common to automatically generate code from informal models (model driven architecture). Similar generative approaches should be done using formal models (and modelling languages).

5. *Increase automation and efficiency* and *integrate automation and interaction*
   Automation is crucial for the usability of verification. However, we know that not everything can be done automatically. Hence, the interaction with the verifying person is of great importance: Feedback on source code level, counter examples in case of an error, hints to where the verification failed if no decision could be made, ... Moreover, tools need to provide good support (e.g., using static analyses) for the search for auxiliary specifications, such as loop invariants, lemmata, modifies clauses.

Another important point which is somewhat orthogonal to the presented points is the *acceptance* of formal methods amongst developers. With formal methods becoming a stable field, they have to become an integral part in the education of computer scientists and software developers. Even if they do not present the formal details, already basic programming courses should teach specification techniques to make them more popular and accepted.

For a better acceptance in development and teaching, tools need to be mature and stable. Expectations towards formal methods are high, and people may turn away discouragedly if these expectations cannot be fulfilled. It is better to introduce a tool which handles a few topics perfectly, rather than to introduce a powerful tool which fails as soon as the input deviates from a schema.