On the Verified-by-Construction Approach

Michael Butler
University of Southampton
17 February 2006

Introduction

At the VSTTE (Verified Software: Theories, Tools, Experiments, vstte.inf.ethz.ch) conference held in ETH Zürich in October 2005, Tony Hoare and Jay Misra presented a vision of an international Grand Challenge to construct a program verifier. This vision appears to be having a very powerful catalysing effect on getting researchers in all manner of formal development approaches to pool resources and work towards more common goals. This is borne out by the large gathering of top researchers at the VSTTE conference and by the subsequent establishment of working groups set up to refine the challenge further. This article is my attempt at making the case for the so-called verification-by-construction approach to formal development and the contribution it can make to the challenge of verified software.

Why verification-by-construction is important

Much discussion on the need for a powerful program verifier seems to contain the following underlying assumptions:

- That a program verifier will be used mostly to verify programs
- That when verification fails it is because the program contains errors

While a powerful program verifier is a very valuable tool for programmers, it does not help them construct a verifiable program in the first place. Equally, the quality of any verification is dependent on the validity of the formal properties against which a program is checked. The verification-by-construction approach helps developers who want to construct reliable software systems by addressing the following questions:

- How do we construct properties against which to verify our software?
- How do we construct our software so that the verification will succeed?

The verification-by-construction approach is about providing design tools that help developers produce reliable software. It broadens the focus away from just being analysis of the finished *product* and addresses better the development *process*.

How can verification-by-construction be achieved?

Verification by construction can be achieved by having a formal framework in which models are constructed at multiple levels of abstraction and related by refinement[1] relations. The highest levels of abstraction are used to express the required behaviour in terms of the problem domain. The closer it is to the problem domain, the easier it is to validate against the informal requirements, i.e., ensure that it is the right specification. The lowest level of abstraction corresponds to an implementation

[1] According to dictionary.com, 'to refine' means 'to reduce to a pure state'. Ironically our use of the term has the exact opposite meaning. The term 'reify' (as used by Cliff Jones and others) is perhaps more appropriate for what we do but is far less widespread. I expect we are stuck with 'refine'.

or to a specification from which an efficient implementation can be derived automatically. Also critical in this framework are mechanisms for composing and decomposing models. Composition can be useful for building up specifications by combining models incorporating different requirements. Decomposition is important for relating system models to architectures of subsystem models and subsequent separate refinement of subsystems.

Ensuring that two models, M1 and M2, are in a refinement relation may be achieved in one of two ways:

1) Posit-and-Prove: The developer provides both M1 and M2 and uses tools to verify that M1 is refined by M2. In some cases this might be possible using a model checker. Alternatively a tool will generate proof obligations which can be verified using powerful theorem provers or possibly checked using model checkers. Typically this approach requires properties such as invariants and variants to be provided by the developers.

2) Transformational Approach: The developer provides M1 and applies a transformation that automatically constructs M2 in a way that guarantees refinement. This might result in the generation of side conditions that will need to be verified but discharging these should be a lot less effort than proving that M1 is refined by M2 in the posit-and-prove way.

One can immediately see how the transformational approach helps developers to construct software such that the verification will succeed. Unfortunately a fully transformational approach for a broad range of problems and solutions is far from being realised so that the posit-and-prove approach will rule for the foreseeable future. It might not appear immediately clear how the posit-and-prove approach helps developers to construct software for which the verification will succeed since the developer is expected to provide M2 as well as M1. This is where having multiple levels of abstraction is important. Typically there is a large abstraction gap between a good formal specification, i.e., one that is easy to validate against the requirements, and an efficient implementation. This gap means it is more difficult to be guided by the specification when constructing an implementation. By having smaller abstraction gaps between a model M1 and its intended refinement M2, it is more natural to be guided by M1 when constructing M2. Typically a refinement step incorporates a design decision about how some effect is achieved or represents an optimisation of the design. With a small abstraction gap, the construction of M2 is driven by both M1 and the desired design decision or optimisation. When the construction of M2 is guided by M1, then the verification that M2 refines M1 is more likely to succeed.

A halfway house between transformational and posit-and prove can be envisaged, where certain *patterns* of model and refinement can be captured and used in the construction of refinements. This is a more pragmatic idea than transformational refinement in that the pattern might not guarantee the correctness of the refinement[2]. Instead M2 would be constructed from M1 by application of a pattern and the correctness of the refinement would be proved in the usual posit-and-prove way. Ideally the pattern should provide much of the ancillary properties (e.g., invariants, tactics) required to complete the proof.

[2] A refinement M2 is correct with respect to some model M1 when M2 refines M1.

Models versus Properties

In a refinement approach one does not necessarily distinguish between properties and models. Essentially we are working with models in a modelling language and the important property to be proved of some model M2 is that it is a refinement of some other model M1. In doing this, we may need ancillary properties like invariants, variants and assertions. Good tools can help us discover these ancillary properties as part of the effort of trying to prove a refinement. So the answer to the question 'what properties should we prove of a model?' is 'those properties that allow us to show that it is a refinement of its abstraction'. For the most abstract models, the important property is that they satisfy the requirements of the problem domain. This is an informal check which can sometimes be aided by ancillary properties. Within a particular framework there may be differing strengths of refinement. A weaker notion might capture the preservation of safety behaviour, while stronger notions might capture preservation of liveness and/or fairness.

With a refinement approach the 'creative' input in a development is a collection of explicit models at different levels of abstraction. The invention of ancillary properties is dictated by the need to prove refinement between these explicit models. From an engineering perspective, I would argue that an explicit model is a fairly natural thing to have to create because one can easily get a feeling of 'completeness' of the model (at a certain level of abstraction). When creating properties rather than models I find it is more difficult to achieve that sense of 'completeness'.

In my experience, refinement is never purely top down from most to least abstract. The reason is that it is difficult to get the abstract model precisely right. One usually starts with an idealistic abstract model because that is easy to define. As refinement proceeds and more architectural and environmental details are addressed it often becomes clearer how the ideal abstract model needs to be modified to reflect reality better. Modifications to some level of abstraction will ripple up and down the refinement chain. This is not a weakness of the refinement approach per se, rather a reflection of the reality of engineering of complex systems.

It goes without saying that the refinement relation should enjoy some form of transitivity. I say 'some form of transitivity' because refinement is based on comparing some notion of what can be observed about a model and it is useful to be able to modify what can be observed at different levels of abstraction. In particular, the interface to a system is usually described abstractly and may need to be made much more concrete at decomposition or implementation levels. In such cases, the observable behaviour is not directly comparable, but needs to be compared via some mapping and transitivity of refinement is via composition of mappings.

Other points in favour of verification-by-construction

The verification-by-construction approach encourages verification of designs and not just verification of programs. From an engineering perspective, it is possible that there is a greater payoff from verifying designs rather than programs. Does it not seem more likely that a design error would have a detrimental impact on system reliability than a programming error?

As well as supporting verification of designs and implementations, good formal modelling languages encourage a rational design process. The use of good abstractions and simple mathematical structures in modelling can lead to cleaner, more rational system architectures that are easier to understand and evolve than

architectures developed using less disciplined approaches. Being able to verify a system is not enough. It is also important to be able to test, maintain and evolve it. This is facilitated by rational design.

The inclusion of annotations such as invariants and assertions in programming languages (e.g., Eiffel, Spark Ada, JML, Spec#), along with associated analysis tools, provide powerful support for programmers. However, this approach is not enough on its own as these annotations are designed to specify properties about programs but do not easily allow for reasoning about the contribution an individual program makes to the overall reliability of a system. Control systems, interactive systems and distributed systems involve multiple agents (users, environments, new programs, legacy components) all of which contribute to the reliability of a system. Individually the agents may be very complex so reasoning about compositions of agents in all their gory detail may be infeasible. Instead, there is evidence that it will be feasible to reason about complex systems through good use of abstraction, refinement and decomposition.

When verifying a program directly one is having to reason about a number of issues simultaneously; the problem to be solved, the data structures used in the solution and the algorithmic structures used in the solution. If these issues can be factored out and dealt with separately as much as possible, the proof obligations can be simplified and the reasoning made more manageable. Abstraction and refinement supports this factorisation. It is often possible to model and reason about how a strategy solves a problem in an abstract way using abstract algorithmic and data structures. This abstract solution can then be optimised by introducing more concrete algorithmic and data structures through refinement. Reasoning about these optimising refinements no longer requires reasoning about the original problem as this will have been dealt with by the earlier refinement. By keeping the models as abstract as possible at each level, we will have simpler proof obligations to discharge. At higher levels of abstraction we focus the reasoning more on the problem domain and less on the details of the particular solution.

Further questions

Which notations should be used? My own experience is that one can go a long way with set theory and logic as used, for example, in Z, VDM and B. Dealing with reactive and distributed systems in these notations requires richer notions of refinement and decomposition, but not necessarily major extensions to the notations. It some cases it is appropriate to augment set theory and logic with notations such as process algebra and temporal logic.

What type of systems should we work on in the grand challenge? I am especially interested in multi-user, distributed systems and in control systems involving an environment and believe these will provide many interesting challenges.

What about the link to programming languages? To some extent, the choice of particular programming language is not so important in the verification-by-construction approach. What matters is that a sound mapping can be made between the lower level abstractions used in verification-by-construction and the constructs of target programming languages. There is however interesting overlap between this mapping and important research in programming language design which tries to improve programming abstractions. In particular I am thinking of:

- Declarative styles of programming
- Atomicity and transactional support for concurrent programming

- Abstractions for structured data (e.g., abstractions of XML messages, abstractions of pointer structures)

Clearly, better programming abstractions will make it easier to bridge the gap between models and programs.

The challenge

To a large extent the required theory to support verification-by-construction already exists. The challenge is to provide a powerful set of tools to support abstraction, refinement and decomposition. In achieving this, we should strive to achieve as much integration as possible and avoid silos. We should also exploit as much of the existing and future advances in theorem proving and model checking as possible, as well as advances in programming language design, program verification and automated program generation. As they evolve, the support tools should be applied to the development of interesting software-based systems. No doubt interesting theoretical advances will be identified and achieved along the way as well.