Generating Programs plus Proofs by Refinement

Douglas R. Smith

Kestrel Institute, Palo Alto, California 94304 USA

1 Technical Approach

We advocate an automated refinement approach to developing programs and their proofs. The approach is partially embodied in the Specware system [6] which has found industrial and government applications. Our view is that the future of software engineering lies in the tight integration of synthesis and analysis processes.

1.1 Specifications

Refinement-oriented development starts with the requirements of the procuring organization. These requirements are typically a mixture of informal and semi-formal notations that reflect the needs of various stakeholders. To provide the basis for a clear contract, the requirements must be formalized into specifications that both the procuring organization (the buyer) and the developer (the seller) can agree to. Specifications can be expressed at a variety of the levels of abstraction. At one extreme a suitable high-level programming language can sometimes serve to express executable specifications. However, an executable specification requires the inclusion of implementation detail that (1) is time-consuming to develop and get right, and (2) might be better left to the developer's discretion. At the other extreme, a property-oriented language (such as a higher-order logic) can be used to prescribe the properties of the intended software with minimal prescription of implementation detail. The solution in Specware is a mixture of logic and high-level programming constructs that provides a wide-spectrum approach, allowing specification writers to choose an appropriate level of abstraction from implementation detail.

1.2 Refinement

A formal specification serves as the central document of the development and evolution process. It is incrementally refined to executable code. A refinement typically embodies a well-defined unit of programming knowledge. Refinements can range from situation-specific/ad-hoc rules, to domain-specific transformations, to domain-independent theories/representations of abstract algorithms, data structures, optimization techniques, software architectures, design patterns, protocol abstractions, and so on. KIDS [9] and Designware [12] are examples of systems that automate the construction of refinements from reusable/abstract

design theories. A crucial feature of a refinement from specification A to specification B is that it preserves the properties and behaviors of A in B, while typically adding more detail in B. This preservation property allows us to compose refinements, meaning that a chain of refinements from an initial specification to a low-level executable specification can be treated as a single property- and behavior-preserving refinement, thereby establishing that the generated code satisfies the initial specification. An intrinsic capability of a refinement process is that proofs of consistency between the source and target of a refinement can be composed in a similar way. While this capability has not been implemented for general-purpose design (to our knowledge), it paves the way for refinement machinery that produces programs and proofs at the same time. Evidence for the feasibility of this capability may be found in several recent domain-specific code generators that generate programs, proofs, and other certification documentation automatically from specifications; e.g. the Specware-based JavaCard Applet generator from Kestrel [3], and the AutoBayes and AutoFilter projects at NASA Ames [14, 15].

Why take a refinement approach to developing programs plus proofs? We can look at this question from several points of view.

1. *Software Lifecycle View* – Boehm observed many years ago that it is exponentially less expensive to fix an error in the requirements phase than to fix it in subsequent design, maintenance, and evolution phases [2]. The lesson is to focus attention on the requirements up-front and to rigorously maintain consistency during development and evolution (for a comparative experiment along these lines see [16]). This is the essence of refinement-oriented development. Boehm also estimated that many measures of software cost (including cost, schedule, and errors) increase superlinearly in code size (approximately to the 1.2 power). By focusing on specifications that omit implementation detail, the developer gains a quantitative advantage in managing this intrinsic complexity during design and evolution. In a simple experiment at Kestrel, Green and Westfold [5] show a 2-5x increase in dependencies when moving from specs to code. Since dependencies seem to be closely related to the inertia of complex systems, again the advantage accrues to a development process that focuses development effort on specifications and early designs.

2. *Language View* – Progress in Computer Science is often measured by progress in the the abstraction levels provided by programming and modeling languages. The natural continuation is towards languages that abstract away increasing amounts of implementation detail, together with the generative mechanisms for adding the missing detail. Compilers and automatic program generators typically work by supplying fixed implementations for the constructs of the source language. Clearly, for general-purpose design there will not be unique ways to implement property-oriented specifications, but instead there will be a choice of design abstractions that may apply. Therefore, a general-purpose refinement environment will be interactive, in much the same way that many successful theorem-provers are currently driven by user-guidance and automated tactics (e.g. PVS, HOL, Isabelle). However, by narrowing the scope to domain-specific

specification languages, the refinement process can again be fully automatic; e.g. Planware [1], JavaCard, AutoBayes, AutoFilter, and others).

3. *Cost of Assurance View* – A refinement-oriented development process has a different cost structure than traditional development. There is a capital investment in building domain models and machine-usable design knowledge, but these costs are amortized through reuse. Thus the cost of producing the first program in a domain may be higher, but the incremental cost of producing subsequent versions and other programs in the domain should decrease below the level of conventional programming. The cost of handling of proof obligations of specs and refinements should be less than the cost of analyzing the final product. Moreover, the incremental cost of (re)assurance under evolutionary steps is dramatically reduced under refinement when compared with post-hoc verification. The key lies in the reuse of the refinements. In a domain-specific setting, an evolution step is performed by modifying the initial specification or model, and then automatically generating a new program, plus its proof and certificates. In a general-purpose setting, the refinement structure may be at least partially replayed. If the refinements are generic and derived from reusable design knowledge, then the refinements may often still apply and absorb the changes. The assurance costs are mainly borne at library-development time when the generic/reusable refinements are verified (or generated correctly).

2 Progress – Specware Foundations

Specware [6] provides a mechanized framework for the composition of specifications and their refinement to codes in several programming languages. The framework is founded on a cocomplete category of specifications. The specification language, called MetaSlang, is based on a higher-order logic with predicate subtypes and extended with a variety of ML-like programming constructs. MetaSlang supports pure property-oriented specifications, as well as executable specifications and mixtures of these two styles. Specification morphisms are used to structure and parameterize specifications, and to refine them. Colimits are used to compose specifications, instantiate parameterized specifications, and construct refinements. Diagrams are used to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification. A recent extension of Specware supports the specification, composition and refinement of behavior through a category of abstract state machines [8].

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, datatype refinements, program optimization rules, software architectures, abstract user interfaces, and so on, are represented as diagrams of specifications and morphisms. We arrange these diagrams into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications [11].

The framework is partially implemented in the research systems Specware, Accord, and Planware. Specware provides basic support for composing specifications and refinements, and generating code. Code generation in Specware is supported by inter-logic morphisms that translate between the specification language/logic and the logic of a particular programming language (e.g. CommonLisp, C, and Java). It is intended to be general-purpose and has found use in industrial settings. Accord extends Specware to support the specification and refinement of behavior and the generation of imperative code [8]. Planware transforms behavioral models of tasks and resources into high-performance scheduling algorithms [1].

A key feature of Kestrel's approach is the automated application of reusable refinements and the automated generation of refinements by instantiation. Previous experience with manually constructed and verified refinement (e.g. in VDM or B) has resulted in small-scale developments and costly rework when requirements change. In contrast, automated construction of refinements allows larger-scale applications and a more rapid evolution process. For example, in the 1990's Kestrel developed a strategic airlift scheduler for the US Air Force that was entirely evolved at the specification level. The application had about 24000 lines of generated code from a first-order logic specification. Over 100 evolutionary derivations were carried out over a period of several years using the KIDS system [9], each derivation consisting of approximately a dozen user design decisions. Current scheduling applications being developed using the newer Planware system consist of over 100k lines of generated code from less than 1000 lines of source specification. Code generation is completely automatic and takes a few minutes.

3 Research Challenges

Although we have emphasized a synthetic approach to program assurance, our view is that the future of software engineering lies in the integration of synthesis and analysis processes. By way of analogy, conventional programming relies on a tight integration of synthesis and analysis – compilers and their type analyzers work together to check the programmer's work and generate executables; and inside the compiler, flow analysis is used to inform various optimizations that transform and speed up the executable.

Similarly, we see an opportunity for a tight integration between the refinement process and the assurance generation process. At the spec level, there are proof obligations on the static consistency that can be verified (this is essentially an extended type analysis that includes for example, checking that predicate subtype properties hold). We would like to record such proofs together with the specification. When we compose specs, we would like to reuse those proofs to obtain necessary proofs of consistency of the composite specification.

Refinements also have proof obligations, which can be handled in a post-hoc verification style, or in a correct-by-construction style (which we prefer). Constructive theorem-proving (e.g. [4, 7]) in the correct-by-construction generation

of refinements provides a clear example of integrated, automated analysis and synthesis. In order to construct an expression to serve as the translation of a domain symbol in a specification morphism, we set up the problem of finding a constructive proof for a forall-exist formula [10]. A witness for the existential may be found by a saturation procedure in general logics or algorithmically by constraint solving in special theories.

One example of refinement construction involves the use of constructive theorem proving and reuse of code templates – code fragments for instantiating a code template can be generated as witnesses to existentially quantified variables in the correctness formula for the template, resulting in a correct-by-construction instantiation of the template [10, 13]. The proving should be more robust and informed because it is taking place in a context where both intent (the specification) and design knowledge (the template and code context) are explicit.

Another point of connection lies in refinement generators. Some optimization techniques, such as simplification, partial evaluation, and finite differencing, are written as metaprograms that use a constructive prover and generate refinements [9]. To support the construction of both code and proofs, the metaprogram must produce also a proof for the obligations of the refinement and appropriately transform the proofs in the source specification.

One general need is for mechanisms to carry proofs along with specs as they undergo refinement. With such mechanisms, the definitions and proofs at one level are refined to definitions and proofs at the next level, so that proofs are integral to design, rather than a side activity. Proofs must also be composed under colimit, so that again, all composition and refinement activities produce both code and proofs.

The discussion above focuses on technical approaches to developing program and proof together during a refinement process. An orthogonal issue is the control of such a refinement process. As noted earlier, for general-purpose design it unlikely that there can be a tractable automatic procedure for translating requirement-level specifications to correct and efficient code. Our research prototypes KIDS [9] and Designware [12] support an interactive interface in which users select which form of design knowledge to apply next (design knowledge is embodied in metaprograms that may invoke constructive inference for subtasks). The Specware system [6] supports user-written metaprograms which, at compile time, have the effect of generating implementations from specifications in higher-order logic. Our experience with developing a variety of medium-scale applications with these systems is that, because of the granularity of the knowledge representations that we use, relatively few user decisions (a dozen or two) are needed to generate fairly complex, but correct, code.

However, the sweet spot for automated code generation lies in narrowing down the specification language and design knowledge to specific application domains. Planware [1] and JavaCard [3] are two domain-specific fully automatic code generators built in Specware. Planware defines a domain-specific requirement language for modeling planning and scheduling problems. From such a problem model (typically 100 to 1000 lines of text derived from mixed text

and graphical input), it automatically generates a complex planner/scheduler together with editors and visual display code (over 100,000 lines of code for some applications). By narrowing the specification language, the generator developers can effectively hard-wire a fixed sequence of design choices (of algorithms, datatype refinements, optimizations) into an automatic design tactic. The JavaCard generator (see paper in this issue) furthermore also generates proof of consistency between specification and code as a by-product of refinement.

4 Concluding Remarks

In light of the above discussion, we see the following activities as key aspects of an effective and scalable technology for generating programs and proofs of their consistency with specifications.

- Developing libraries of design theories and refinement generators – e.g. theories and application mechanisms for generating refinements that embody knowledge of architectures, formalized design patterns, policy classes, algorithm theories, datatype refinements, optimization transforms, and so on.
- An elaborated refinement process that tightly integrates the generation of refinements with recording, composition, and refinement of proofs.
- Experiments in the rational reconstruction of benchmarks – develop formal specifications for various benchmarks, as well as relevant domain theories and design theories, followed by a rational reconstruction of the design by refinement of similar code from the specifications.

References

1. Becker, M., Gilham, L., and Smith, D. R. Planware II: Synthesis of schedulers for complex resource systems. Tech. rep., Kestrel Technology, 2003.
2. Boehm, B. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J., 1981.
3. Coglio, A. Toward automatic generation of provably correct Java Card applets. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs* (July 2003).
4. Green, C. Application of theorem proving to problem solving. In *Proceedings of the First International Joint Conference on Artificial Intelligence* (1969), pp. 219–239.
5. Green, C., and Westfold, S. Experiments suggest high level formal models and automated code synthesis significantly increase dependability. Tech. Rep. KES.U.00.8, Kestrel Institute, January 2001.
6. Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.
7. Manna, Z., and Waldinger, R. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems 2*, 1 (January 1980), 90–121.
8. Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering* (2001), IEEE Computer Society Press, pp. 157–165.
9. Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.
10. Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.
11. Smith, D. R. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.
12. Smith, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.
13. Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.
14. Whalen, M., Schumann, J., and Fischer, B. Synthesizing certified code. In *Proc. Formal Methods Europe (FME 2002)* (Copenhagen, Denmark, 2002), Springer LNCS 2391, pp. 431–450.
15. Whittle, J., and Schumann, J. Automating the implementation of Kalman filter algorithms. Tech. rep., NASA Ames Automated Software Engineering Group, 2004. submitted for publication.
16. Widmaier, J., Schmidts, C., and Huang, X. Producing more reliable software: Mature software engineering process vs. state-of-the-art technology? In *Proceedings of the International Conference on Software Engineering 2000* (Limerick, Ireland, 2000), ACM, pp. 87–92.