Derivational Software Engineering

Douglas R. Smith
Kestrel Institute
Palo Alto, California
smith@kestrel.edu

Louis Hoebel
GE Global Research
Niskayuna, New York
hoebel@ge.com

Categories and Subject Descriptors

D.1.2 [**Software**]: Automatic Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification
; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design

1. Current Software Practice

Software exists to fulfill needs that individuals and organizations have. Software is a complex artifact that fulfills those needs by providing services, while consuming reasonable levels of resource and conforming to constraints from its context. To develop software, developers bring their design knowledge to bear, translating their understanding of the requirements by adapting existing code and creating new specialized code. Unfortunately, the design knowledge and its rationale are usually not captured in a useful form.

In our view, software should be treated as a formal composition of requirement specifications, models, library components, design abstractions (e.g. system architecture patterns, design patterns, algorithm patterns, etc.), datatype refinements, optimizations, and other specialized code generation techniques. We call this composition the *derivation structure* of the software. Our intent is that a machine could execute a derivation structure to generate code, effectively mechanically replaying a summary of the developer's design process.

Current practice leaves the derivation structure informal and largely unrecorded, giving rise to critical deficiencies:

- *Cost of evolution* – Most of the cost of software over its lifecycle results from adapting the code to meet changing requirements. Without an explicit derivation structure, there can be little in the way of machine support for software evolution.
- *Lack of confidence* – The design and evolution process leaves behind little or no evidence that the artifact achieves its goals. How do I as a buyer know that a product has the functionality that it claims? How do I know that it operates safely and securely in my environment? What evidence can I examine to determine the validity of the advertised features of the product?
- *Cost of certification* – Because of the informal development process, the cost of certifying software systems is often several times the cost of developing the software. It would be far better if the development process itself generated verifiable (i.e. independently and efficiently checkable) evidence that the code meets its requirements. For example, if a design pattern were formalized, and pre-analyzed for its properties, then it would be possible to generate certification evidence at the same time that the code pattern is instantiated.

To address these deficiencies, a science of design providing solid foundations for software engineering will focus on derivation structure as the essence of software. A derivation structure provides better localization and modularization of concerns than code does. Software is more than source/binary code, *software is its derivation structure.*

2. Derivational Software Engineering

Software exists to fulfill needs that individuals and organizations have. As these needs evolve, they are reflected in changing requirements on software systems. Some changes are purely technological, as in the need to migrate to newer platforms, but most reflect the need for new or modified features and services. Needs also arise to deal with changing features of the operating environment of software, such as the increasing presence of malware and non-benign users. Overall, *changing requirements are the driver of the software development and evolution process.*

In this view, the software lifecycle is primarily defined by a sequence of requirements. Secondarily, each requirement has an associated derivation structure. The design decisions taken in the development of code for an initial requirement for a system would be recorded as its derivation structure. Subsequently, an incremental change to the requirements would trigger the need for an incremental change to the derivation structure, which includes changed code. The challenge and promise of a derivational approach is to treat incremental change to a derivation structure as a problem-solving process admitting automated solutions, leading to increased automation and lower cost due to reuse of the previous derivation structure. The issue is knowing when

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

to follow the previous derivation structure and when to depart from it, generating new structure. In summary, the initial system design derives from the initial requirements, and subsequent evolution steps are driven by requirements changes.

Requirement specifications capture the conditions under which stakeholders find a candidate implementation acceptable. The requirements community has developed a rich collection of techniques for composing requirements, exploring tradeoffs, and moving towards formalization as specifications [24]. Requirements may be best expressed by a mixture of (1) models that capture features and properties of the system (e.g. of physical agents and domain-specific requirements), and (2) logical formulas that decide the acceptable behaviors of the system. The models define an overapproximation (i.e. superset) of the desired system behaviors, and the logical conditions serve to eliminate unacceptable behaviors. The most familiar logical requirements are safety and liveness properties, which are expressible as predicates on traces. Qualitative (i.e. nonfunctional) requirements are often given by preferences over coarse-grain utility measures on traces.

In a derivational approach, it is not necessary that logical requirements are ever complete. First, the incremental adaptation of derivation structure to changing requirements allows for a process that starts with incomplete requirements and slowly "trains" the development process/tools by incrementally adding in new requirements. That is, in a requirements-driven process, it may be better to incrementally develop the requirements, so that with each new requirement increment, the developers work on the corresponding increment to the derivation structure. Second, models often provide a more concise means for capturing requirements and given infrastructure than through logical formulas. The issue is ensuring that the features and properties embodied in models are preserved by the derivation in the final code.

It should be obvious that all code embodies some requirements, whether or not they reflect actual needs. And it is also true that feedback from stakeholders is needed to capture the actual requirements. The initial requirements may be a crude approximation of what they will end up as. The goal is to develop a succession of approximations that converge to requirements that all stakeholders find acceptable. Over the lifecycle, as the needs of stakeholders evolve, so will the requirments. The software lifecycle is a sequence of requirements that are more-or-less acceptable to the stakeholders. The job of software engineers is to supply the tools and design-knowledge formalizations that permit an automated, incremental generation of derivation structures accompanying this sequence of approximate requirements. If we regard the code generated from intermediate requirements as prototypes, then the prototypes can be run for stakeholders to gain further insight into the adequacy of the requirements, unconcerned about the inertial cost of modifying the prototype (since it and its successors will be generated).

In our view, the process of transforming requirement specifications to code is via a sequence of refinement steps. Each refinement embodies a design decision, replacing a design on one level (initially the combined models and formulas that comprise the requirement specifications) with a more detailed design at the next. Under certain mathematical conditions, refinements can be shown to preserve properties. The key to generating certification evidence as a part of the code derivation process is to ensure that the refinements in the derivation structure (1) preserve the features and properties embodied in initial models, and in the designs at various levels of refinement, and (2) enforce and preserve any logically specified requirements during refinement.

Aspect-oriented and feature-oriented programming provide some rough special cases of derivational design: modular statements of requirement intent are mechanically translated into code scattered throughout an intermediate-level design [2, 5, 14, 23].

3. Directions for Future Research

Several key research issues arise from the view of software as a refinement-based derivation structure in a requirements-driven lifecycle.

1. *Filling the semantic gap between requirements and code* – Where does the information come from to fill the large gap between requirements and code? What is the nature of the refinement steps and how do they arise?

   Compilers fill in their gap by instantiating rules for translating constructs from one language to another. Requirement-level specifications often do not have computational content so there may not be a ready set of source-to-source translation rules. Some refinement-based methods such as VDM, B [1], and the Praxis' Spark methodology [4] rely on manually invented refinements and their post-hoc verification. This is an expensive process under requirements evolution.

   In our view, most software can be treated as the routine composition of well-known *design abstractions*, as described in textbooks on system architectures, design patterns, algorithms, data structures, and so on. Capturing those *reusable* design abstractions and making them available for machine application will facilitate the construction (and incremental reconstruction) of derivation structures.

   The informal capture of design abstractions is not a novel or little-known idea. Software architecture patterns [8], design patterns [13], frameworks, generic programming [3], template/schema-based programming, generative programming [11], higher-order functions, domain-specific generators, product-lines [10], libraries, etc. all are motivated by a similar objective: to capture common patterns of computation so that they can be built once and reused often [7]. The cost of building them is amortized over many uses. However, the abstraction mechanisms listed, typically are not represented in a way that they can be applied mechanically. Similarly, they do not provide much support for their correctness argument; design abstractions are most commonly thought of as a means for improving programmer productivity and facilitating requirement changes.

   The formal capture of design abstractions has seen less effort, but it naturally builds on and extends the previously mentioned approaches. The benefits of formalizing design abstractions include mechanical application and the co-generation of code and certification

evidence. Work on the formalization of algorithm theories, datatype refinements, and various optimization techniques can be found in [15, 9, 19], including a taxonomy of algorithm theories [20] and the beginnings of a taxonomy of system architecture theories [22]. The creative effort is to develop design theories that are (1) abstract enough to cover a range of concrete design problems, yet (2) structured enough to admit tractable machine support for instantiation.

Domain-specific generators that are based on codified design abstractions and that transform user-specified problem requirements into certifiable code provide viable instances of this approach [6, 12, 16, 17].

Most approaches to code reuse aim to separate common code/structure from variation points (code libraries are the extreme case where there is little or no variation). Attaching properties to the common structure, and attaching constraints on variation points provides a first step towards supporting the generation of certification evidence at the same time that code instances are generated.

The upshot is the need for a research program of codifying best-practice design abstractions in such a way that (1) they can be efficiently instantiated to meet requirement-level specifications, and (2) evidence for the consistency between the specifications and the instance can be co-generated and then efficiently checked by an external observer.

2. *Requirements-driven change* – What kinds of tools can support the adaptation of a derivation structure to a requirements change? In a derivational approach, evolution is effected by a local change to requirements, followed by a process of propagating the change through the derivation structure, re-establishing a consistent refinement chain from top (requirements) to bottom (code). Detection and treatment of inconsistencies between requirements or between requirements and previous design decisions motivate the exploration of tradeoffs and the choice of alternative design abstractions. Significant research is needed to develop both theoretical foundations and practical methods for re-establishing a globally consistent derivation structure after change to a local part. Significant levels of automation for this process seem possible.

   This is a topic that has received scant attention, partly because well-developed machine support for derivation is needed before the issue of incremental rederivation comes to the foreground.

3. *Automated inference tools to instantiate design abstractions* – Some form of deductive inference is needed to specialize design abstractions to requirements and context in a way that generates formal evidence of consistency. Inference tools are needed to calculate instances of design abstractions that are correct-by-construction and tailored to context.

   The inference techniques underlying model checking, abstract interpretation, and other forms of theorem-proving and analysis, provide sound methods for reasoning about the properties and behaviors of software. Work on derivational approaches to software has traditionally built on and extended techniques used for post-hoc verification of code. Derivational software engineering has characteristic needs beyond those of program verification, especially (i) constructive techniques for finding witnesses to existentials, (ii) constraint solvers, and(iii) methods for propagating properties through models and patterns. Current efforts to develop fast specialized inference engines (SAT, SMT, combined decision procedures) are especially important in providing efficient automatic support for constructing refinements.

4. *Generation of certification evidence* – A formalized design abstraction can be associated with an abstracted proof (or other form of evidence). The inference tools constructing a refinement from the abstraction can then also generate formal evidence for the refinement; that is, evidence of the consistency between the source and target of the refinement [21]. Evidence for the consistency of each refinement can be composed in a natural way in order to provide evidence for the consistency of a refinement chain [21].

The derivational approach to software provides fresh avenues of research for established subareas of Computer Science. It offers to provide integration opportunities between communities that tend to have little interaction and who should be seen as parts of a larger whole, including

- Requirements engineering
- Formal specification languages and logics
- Domain-specific languages and models
- System architecture patterns and modeling formalisms
- Design patterns
- Model-Driven Development
- Formal refinement methodology
- Algorithms, data structures, optimizations
- Deductive inference, combined decision-procedures and constraint-solvers
- EDA (hardware synthesis tools)
- Resource mapping and optimization

For example, increased interaction between the requirements, specification, and system architectures communities would seem to be natural. Current research in architectures is mostly oriented toward modeling and verification. To support formal derivation, research is needed into techniques for propagating (global) requirement constraints through the structure of the architecture so as to infer (local) constraints on components. This requires an integrated understanding of requirements, requirement specifications, formal architecture patterns/theories, efficient deductive propagation mechanisms, and so on. Formal capture of architecture patterns and propagation techniques for refining them may require advances in theory. Currently there is a large gap between the basic relevant theory (e.g. coalgebraic/coinductive structure, concurrent games, game logic, mechanism design),

and the conceptions of software engineers regarding the system-level patterns that constitute best-practice design; e.g. [8, 13, 18].

4. Concluding Remarks

Software exists to fulfill needs that individuals and organizations have. An ultimate end of software engineering is providing the tooling to meet those needs through an automated requirements-driven process of creating and evolving software. It may be increasingly true that "most software creators are not software professionals". Software engineers then become the meta-engineers that create the tooling to allow "software creators" to capture and evolve their requirements and automatically generate certifiable software-based services, without needing to know the details of the underlying derivation structures. The design abstractions necessary to provide that level of automation become the focus of software engineer's efforts. Our view is that a skilled software engineer's insights and creativity are best captured in reusable design abstractions. There is higher leverage to be obtained from the effort to develop reusable design abstractions than from serially creating ad-hoc codes. This forecasts a shift, as more developers move from programming to achieve greater leverage through design knowledge capture and meta-programming.

1. REFERENCES

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *AMAST-08, Springer LNCS 5140*, pages 36–50, 2008.

[3] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming – an introduction. In S. D. Swierstra, editor, *Advanced Functional Programming*, pages 28–115. Springer-Verlag, LNCS 1608, Berlin, 1999.

[4] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, Boston, MA, USA, 2003.

[5] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 702–703. IEEE Computer Society, 2004.

[6] M. Becker, L. Gilham, and D. R. Smith. Planware II: Synthesis of schedulers for complex resource systems. Technical report, Kestrel Technology, 2003.

[7] J. Bruno, M. Kinstrey, and L. Hoebel. Common services framework. In *Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT2010)*. Springer, 2010.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

[9] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.

[10] P. Clements and L. M. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Pub Co, Reading, MA, 2000.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley., Boston, 2000.

[12] B. Fischer and J. Schumann. Generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(3):483–508, 2003.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[14] G. Kiczales and et al. An Overview of AspectJ. In *Proc. ECOOP, LNCS 2072, Springer-Verlag*, pages 327–353, 2001.

[15] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6):1–38, 2009.

[16] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Intl. Symp. on Methodologies for Intelligent Systems*, pages 326–335, 1994.

[17] M. Püschel and et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[18] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, 1996.

[19] D. R. Smith. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering*, 16(9):1024–1043, 1990.

[20] D. R. Smith. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)*, volume LNCS 1101, pages 62–84. Springer-Verlag, 1996.

[21] D. R. Smith. Generating programs plus proofs by refinement. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, pages 182–188. Springer-Verlag LNCS 4171, 2008.

[22] D. R. Smith. Calculating refinements in algorithm and system design. Technical report, Kestrel Institute, ftp://ftp.kestrel.edu/pub/papers/smith/mr.pdf, May 2009.

[23] S. Trujillo, D. Batory, , and O. Diaz. Feature oriented model driven development: A case study for portlets. In *Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE-07)*, pages 36–50, 2007.

[24] A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.