Breaking the Software Development Roadblock: Continuous Software Enhancement By Design Maintenance

Ira D. Baxter
Semantic Designs, Inc.
www.semdesigns.com
idbaxter@semdesigns.com

Abstract

The software development paradigm propounded by Semantic Designs, Inc. envisions a design-centric perspective rather than today's all too prevalent code-centric viewpoint. The Design Maintenance vision mandates notions of design and designing that are both formal and practicable. Our notion of a formal design entails three interrelated parts: *specification* (the artifact's functionality and performance goals); *realization* (including architectural design choices and ultimately, code); and *rationale* (justifying the realization of the specification). By practicable, we mean that each of the three parts of our formal notion of design are: 1) manipulable by both machine and software engineer; 2) scaleable for real, industrial-size systems (10's of MSLOC); 3) able to accommodate systems that are realized using multiple domain specific languages (ranging from high-level specification languages to implementations comprised of multiple target-execution languages); and 4) capable of continuous, incremental enhancement and extension.

The implementation of this vision is called the Design Maintenance System™ or DMS™. The current release is DMS 1.0 Reengineering Toolkit, which provides infrastructure for Domain Specific Language (DSL) engineering as well as language neutral components that enable construction of large-scale, mixed-language specification analysis, transformation and synthesis tools. Original research and development for DMS was funded by an award from the Department of Commerce, NIST ATP under the Component-Based Software Initiative (NIST Cooperative Agreement Number 70NANB5H1165).

The Design Maintenance vision and DMS together form the basis for our full, frontal assault on software engineering's challenging areas of "New Software Development Paradigms" and "Software for the Real World". We claim the Design Maintenance vision and DMS together provide a solid foundation for building a formal, repeatable software engineering discipline and practice. In a DMS-based environment, hard won software engineering synthesis knowledge is made reusable through the incremental accretion of an ever-increasing repository of domain-specific transformations and domain-specific languages. Thus, software engineers reusing such knowledge become more effective in that the artifacts are produced using tested language definitions and transformations. Maintenance productivity will be enhanced as well, since the full-fledged implementation of DMS will support incremental, continuous design modification by capturing and preserving instances of the DMS notion of design—specifications, realizations, rationales. Moreover the leverage afforded by DMS in the identification and disposition of design changes on industrial scale designs promise to yield a 50% reduction in change cycle time while eliminating the software error re-injection rate—"collateral damage" or unintentional consequences of change installation. However, critical technical barriers must be surmounted on the path to achieving these productivity and quality breakthroughs. Some of these barriers include: scalable design capture capabilities; semantic reasoning across DSL boundaries; specification of and reasoning about performance-related aspects for a software artifact.

Keywords

Software Engineering, Design, Maintenance, Program Transformation

1 Design *must* be the Key Engineering Product

Traditional software construction methods suffer from high development and maintenance costs as well as long delays. The resulting products often have a large number of errors, and are difficult to adapt to change over their lifespan. The fundamental cause of these problems is the loss of *mechanically*-processable design information about the software, from its specifications, to its architecture, algorithms and implementation, especially including rationale as to why the software is structured/implemented the way it is. Without such design information, software engineers are doomed to waste considerable time re-inventing poor approximations of the true design, and consequently organizations repeatedly pay high (re)engineering costs and compound implementation errors.

Retaining design information by informal means such as documents or human-based system expertise does not allow fast or reliable transfer of that information to other engineering agents or automated tools. The solution to this is to mechanically capture and store system design information as designers work. To ensure that the design is accurate, the resulting system must be derived from the design, rather than informally coding from a design and later attempting to reconcile the as-implemented code with the proposed design. Capturing such design information in a mechanically processable form enables automation to be used during design and implementation, bringing the productivity of computing to the software engineering process.

Automated processing of design knowledge falls in two categories: analysis of a design, to identify consequences, strengths and weaknesses, and semi-automated code generation, to provide accelerated means for reliable implementation. Domain specific program transformation systems are a maturing technology that can provide a significant amount the necessary automation.

Finally, the engineering process must be viewed as continuous product enhancement by incremental design maintenance. Even "new" systems are designed and implemented incrementally over time. This paper offers a vision of Design Maintenance, sketched originally in [2,3,5], and outlines DMS, a tool under construction, which is intended to lead, in the long term, to a realization of the vision.

2 What is a *Design?*

The idea of capturing "designs" is certainly not new, and a plethora of CASE methodologies and tools have been proposed, implemented and used in the past. We contend that their payoff would have been much greater with a *commitment to a precise definition of design*. The failing of many CASE tools is caused by commitment to extremely weak notions of design.

The value of design information is in the set of questions it can potentially answer about the designed artifact. We argue that a good definition of design must provide sufficient information to, in principle, answer *all* possible questions about the designed product, including What, How, and Why:

- What is the specification for the software, both:
  - functionality and
  - performance (e.g., constraints concerning environment, capacity, responsiveness, etc.)
- How the software is realized
  - "architectural" design choices
  - final code
- Why the realization meets the specification

Finally, such design information must be extensible thus admitting to answering new classes of questions as software engineering research and practice is able to formulate them. Each of the following subsections elaborates these components a design.

2.1 Domain-oriented Specifications

The first component of a design is the *specification* of the software constructed. This answers the fundamental question of "What should the software do, and how well should it do it?" Such specifications must be formal, so that there can be no question about intent. They must also be complete; "specifications" which elide some desired behavior are not specifications, but merely projections of the intent. One cannot implement a perspective. We insist that a good design representation be able to specify the required functionality and performance completely, without over committing, to leave maximal room for implementation choices.

This opens the question of which *single* specific formalism should one choose for specification. A standard response to this question from much of the formal methods community is some variation of a mathematically based system involving quantifiers, sets, and relations (such a Z), on the grounds that all computations can in principle be expressed in this way. We hold that the question is malformed as it is predicated on the assumption that there can be a one-size-fits-all formalism.

Instead, we suggest that a software system specification should use *multiple, domain-specific languages* (DSLs) appropriate to the task at hand.[1] Using a DSL offers

[1] We observe that this idea already actually appears in the implementation of practically every software system of any

brevity and clarity, by using the vocabulary of the problem domain. The practical utility of individual DSLs is shown by their variety [10]. DSLs enable specifications to be more easily encoded and reviewed, minimizing errors in encoding requirements. Since fixing requirements errors cost 10-100 times the cost of fixing coding errors, it makes sense to maximize the clarity of the actual system specification.

Each DSL must have a formal meaning. This is a necessity if there is to ever be any mechanical tool support, and if the "intuitive" meaning of the DSL, evident to the designer, is to make any sense. That formality may itself be expressed in terms of underlying but more abstract DSLs, finally grounding in algebras, the lambda calculus, or models. These underlying DSLs can provide the means for a mechanical system to reason about the user-level DSLs.

Complex software systems must almost always handle complex situations, which is unlikely to be covered by a single DSL. By using *multiple* DSLs, each appropriate to a particular aspect of the problem, one can express the overall functionality. For example, one could express a problem involving the specification of discrete control for factory automation in terms of hierarchical, Colored Petri nets [8] propagating values whose arithmetic is defined by multiple algebras (e.g., Boolean, Integer, and Time).

Use of multiple DSLs enables us to express the functionality with one set of DSLs, and the performance properties of the system using another set of DSLs. This could enable us to express performance of algorithms using a DSL for computational complexity, and performance specifications for storage in terms of a temporal calculus describing an eventual steady state. As other definitions of performance arise, appropriate DSLs can be designed for their specification. Performance DSLs are useful when there is a partial ordering over performance expressions, as this allows the comparison and selection of "better" performing implementations [2].

Finally, use of multiple DSLs is robust in the face of the impossibility of having the all-encompassing wide-spectrum language already designed in advance and suitable for every possible application. We can always fall back on whichever (or even many) of the mathematical specification languages seems best, as just another DSL.

A significant research challenge for this approach is the requirement to define the formal meaning of composite specifications coded in multiple DSLs. It requires that composing DSLs must also have a formal meaning [9] (including formally defining certain compositions as nonsense). We hold that the expression and meaning of such compositions is itself a kind of DSL. The mathematical machinery for defining relations and therefore, compositions between the formal meanings of problem domains is beginning to appear in research on semantics with one promising area of inquiry being institution morphisms [12].

scale. Typical examples include COBOL plus SQL, C++ and IDL. We see this at the specification level in UML with OCL, etc.

2.2 Software Realization

The second component of software design is its *realization*, both in the concrete (i.e., the "executable source code") and in the abstract (how the software is mapped from specification into code). This answers the question, "How is the software implemented to achieve correct functionality and satisfactory performance with respect to the specification?"

Realizations are recursive decompositions of subsets of the functionality specification, in terms of lower-level specifications and functionality, usually in mixed target execution languages—"DSLs" such as C++, Prolog, HTML and Verilog. Consider again the example for factory automation. Places in the Colored Petri Net specification are realized by abstract sets of algebraic values, which in turn are realized by C++ arrays of structs representing individual values. Transitions in the Colored Petri Net are realized by predicate-triggered atomic transactions inspecting transition input states, and updating output states; the atomic transactions are realized in turn by state-update-calls to interrupt-disabled C++ code to move values from input place arrays to output place arrays, performing combinational arithmetic along the way.

The actual realization of a specification fragment in terms of lower level constructs is a design choice. This design choice would be recorded explicitly in the design. Such a record is fundamental to enable subsequent incremental design modification. As well it provides bi-directional traceability between the motivating specification fragment and its realization. This traceability will also allow later revalidating the design choice as legitimate if necessary, by informal inspection by other software engineers, or by automated proof based on the semantics of the specification fragments being realized, and the realizing implementation fragments. We remark that other design choices are typically possible and latent in the specification. The potential for other design choices enables redesign of the software during later maintenance.

The realization may be as complete as the stage of the implementation permits, from empty (no design work has been done) to fully realized as source code. Realization parts may be more or less complete than other parts. This allows an engineer to operate on any part of the design, either to extend it in detail towards an implementation, or to retract implementation steps, abstracting a portion of the software to be re-implemented with another technology [1]. Of course, a completed design has "code" components at all "leaves", and may thus be compiled to an executable system.

The part of the realization closer to the specification, that shows how the code is structured at a high level, would be generally termed *architecture* of the code. We view architecture as more fluid than others might; it is necessary to explain the structure of the code, but is itself just a consequence of higher-level design decisions that chose that architecture rather than some other[2].

2.3 Rationale for Realization

The third fundamental component of a design is the *rationale* justifying each design decision. The rationale explains, explicitly or implicitly, why the realization is correct from both functionality and performance points of view.

The correctness of the functionality relation between the specification and the implementation fragments can be validated by theorem proving over the semantics of both. This would be an implicit rationale. Actually carrying out the theorem-proving step would constitute "recovering" the justification, and can be arbitrarily expensive and therefore should be avoided.[3] An indirect but explicit validation would refer to an already-proven theorem about the relationship of an abstraction of the specification fragments and of the realization. Such a theorem implicitly constitutes a correctness-preserving program transformation, and we can store the program transformation in a library to act as the validation [2,3]. The notion of a transformation as a representation for the theorem motivates the use of program transformation systems as the automation engine underpinning Design Maintenance.

The validation of the implementation step must verify that the functionality is properly preserved *and* that the resulting fragment, ultimately code, achieves the specified *performance* properties. Performance properties are typically emergent from larger components, and are thus secondary properties, achieved by choosing a realization that satisfies the desired performance characteristics. When the choice is made, one is obliged to assert or prove that the desired performance is achieved. The assertion or proof that it does so can be recorded to avoid later need to re-prove [2,3]. To the extent practical, performance properties can be recorded with the transformation's definition to make evaluation more tractable. One can also use goal-directed performance specification decomposition as a record.

[2] For us, architecture as a term better reflects those design decisions that are difficult to change, because much of the rest of the system depends on the architectural choices, leading to massive revision if changed.

[3] In fact, if we were willing to do this, we would only need the specification, and could simply regenerate the code by theorem proving methods after every specification change.

3 Acquiring Designs

We have outlined a formal design representation consisting of specification, realization, and rationale. Such a notion for design enables answering hard questions about the resulting software artifact: What does it do? How well does it perform? How is it implemented? Why is the implementation correct?

How can we acquire and use such designs to enhance productivity? The "pure" method suggests constructing a design from scratch. That is, from a specification find realizations that are rational. A practical method suggests these could be constructed for legacy software systems using reverse engineering to approximate the original design.

The pure approach can be implemented by manual engineering, much as present day software is laboriously hand-coded. Candidates for implementing specification fragments are enumerated, evaluated, and one is chosen, with the rationale for the choice made being encoded at the same time. The pure approach can also be implemented by using a program transformation system to select from a database of predefined transformations. In practice, we expect a mixed initiative in which transformation engines propose and engineers select, or engineers enrich the database of transformations with new transformations during implementation.

Having a database of transformations would enable a pattern-directed reverse engineering of code for which the design does not exist [4]. Transforms matching some fragment of the implementation would be proposed from the database of existing transformations. Each proposed transform implies a proposed specification being implemented. The engineer would validate and select from among the proposed transforms along with implied specifications and this would be recorded. Alternatively, the engineer would add new transforms as necessary to enrich the database of transforms.

4 Continuous Design Maintenance

A simple model of design use would be big-bang implementation and design consultation only for understanding. The vision we propose is *continuous design revision* by repeated incremental design cauterization and implementation.

This process occurs in a cyclic fashion. In the first step, an undesired property of the system is identified, either in terms of function or performance. The difference between the desired behavior and the actual is encoded as a formal *maintenance delta*. This delta is used to determine the parts of the design that cannot be retained. These parts are removed, leaving a partially implemented design that is consistent with the new desired behavior. Finally, the partial design is extended to a complete design and implementation. The theory for maintenance deltas and incremental design revisions is provided in [2,3].

Productivity is obtained in numerous areas. Specification in terms of changes is both natural and succinct, and is separated from implementation issues. Estimation of the impact of a proposed change can be determined by tracing the downstream affect of the change in terms of what must be removed. Summing previous engineering costs on similar low-level specifications corresponding to the removal points can approximate estimated implementation costs. Actual removal can be automated, and implementations can be augmented by transformation systems whose power grows incrementally over time as engineers add new DSLs or transformations. Finally, multiple engineers can operate on the design concurrently if design changes are treated as long-running transactions.

5 DMS 1.0 Status

Design Maintenance is more than just a vision. Semantic Designs (the author's institution) is committed to implementing this vision, recognizing that it is a long term process requiring additional research in design capture.

The foundation for this implementation is DMS™, a scalable, industrial strength program transformation system (www.semdesigns.com/Products/DMS/DMSToolkit). DMS operates on multiple domains, both low level (C++, COBOL, XML) and high level (SPECTRUM algebraic specifications [7]). DMS is designed for large-scale applications with millions of lines of code. DMS is implemented in a parallel language, PARLANSE[11] to provide as much possible computing power to the problem of symbolic analysis and synthesis.

DMS has been deployed to carry out large-scale automated tasks such as duplicate code detection [6], translating JOVIAL to C, test coverage analysis and as an embedded, re-targetable code generator for industrial automation control.

6 Conclusion

This white paper has sketched a new model of software engineering, based on continuous design maintenance of a formal design consisting of specification, realization, and rationale [2,3]. Industrial-strength technology for implementing foundational components for our notion of design has been constructed and fielded.

References

[1] G. Arango, I. Baxter, C. Pidgeon, P. Freeman, "TMM: Software Maintenance by Transformation", *IEEE Software* 3(3), May 1986, pp. 27-39

[2] I. Baxter, "Transformational Maintenance by Reuse of Design Histories" Ph.D. Thesis, Information and Computer Science Department, University of California at Irvine, Nov. 1990, TR 90-36.

[3] I. Baxter, "Design Maintenance Systems", *Comm. of the ACM 35(4)*, ACM, April 1992.

[4] I. Baxter and M. Mehlich, "Reverse Engineering is Reverse Forward Engineering". *4th Working Conference on Reverse Engineering*, IEEE, 1997.

[5] I. Baxter and C. Pidgeon, "Software Change Through Design Maintenance". *Proc. International Conference on Software Maintenance*, IEEE, 1997.

[6] I. Baxter, et. al., "Clone Detection Using Abstract Syntax Trees", Proc. *International Conference on Software Maintenance*, IEEE, 1998.

[7] M. Broy, et. al, "The Requirement and Design Specification Language Spectrum, An Informal Introduction (V 1.0), Part 1 & 2", Technical University Munich, TUM -I9312, 1993.

[8] K. Jensen and G. Rozenberg (ed)., "High –Level Petri Nets" Springer-Verlag 1991.

[9] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, "Composing Domain-Specific Design Environments", *Computer*, IEEE, November 2001.

[10] A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography", *SIGPLAN Notices*, vol. 35 no. 6 ACM, 2000.

[11] *PARLANSE Reference Manual*, Semantic Designs, 1998.

[12] J.A. Goguen and R.M Burstall, "Introducing Institutions," *Logics of Programming Workshop,* Lecture Notes in Computer Science 164, Springer-Verlag, New York, 1984, pp. 221-225.