Software Productivity through Automation and Design Knowledge

Cordell Green, Dusko Pavlovic and Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304

Abstract

Dramatic increases in software productivity can be brought about by (re)use of software design knowledge and automated composition and refinement. There is evidence that a key reason for the complexity and inertia of large-scale systems is that the number of dependencies in code grows superlinearly with code size. Productivity can be increased by working with software parts that are relatively free of dependencies. This implies the use of modularity and abstraction. We advocate the development of software systems from specifications (formal statements of functional and quality-of-service requirements) and design factors (representations of abstract design knowledge) by means of automated composition, refinement, and analysis.

1 Automating Software Development

Software productivity is affected by many important factors, including design automation, programming languages, methodology, standards, education, process management techniques, and others. In this note we focus on one particular factor: dramatic increases in productivity while maintaining software quality will be achieved through greater *automation*, and that explicit *software design knowledge* is the key to automating software development.

Software design knowledge comes in a variety of forms, but the greatest utility comes when it is explicit (formal), abstract, and machine usable. Lots of useful design knowledge is not in machine-usable form; e.g. in textbooks on algorithms, data structures, design patterns, system architectures, and so on. In particular, the design patterns movement gets at the issue of capturing best-practice solutions to common design problems, but not yet in machine usable form. Some design knowledge is abstract and in machine-usable form, but not explicit. In particular, many software generators (e.g. compilers, model-based generators, domain-specific generators) embody design knowledge about translating models and languages, but often in a form that is not amenable to inference and composition tools (with a few notable exceptions, such as Amphion and Planware).

Abstract software design knowledge, or *design factors*, can be applied to many different requirements, resulting in its embodiment in a variety of applications. Program transformations are a classical example. Various commercially produced tools such as IP (Microsoft), Refine (Reasoning Systems), DMS (Semantic Design), CPS (Grammatech), support design processes based on specification and code transformation. Kestrel researchers have developed formal design theories for a variety of classes of algorithms, data structures, and optimization techniques. In the formal world, there is a long history of projects focusing on the extraction of programs from constructive proofs that their specifications are satisfiable. The proof rules in constructive logic can be thought of as design factors that correspond to the control constructs of a programming language.

We use the term "synthesis" to denote a design process based on automated application of design knowledge. A synthetic approach to software development promises to dramatically improve programmer productivity while assuring that required properties are met. Synthetic approaches focus less on the system code than on the process that composes the code from requirements, design factors, and incremental analysis. That is, the software is treated as a composition of modular statements of requirements and design factors, with the executable code just the outcome of the composition process.

In particular, the synthesis approach starts with a specification of desired properties and refines it to code in a manner that assures that the resulting artifact satisfies the required properties by construction (more generally, with a tunable degree of assurance from the construction process). Each refinement step corresponds to the composition of a design factor with the developing design.

A software synthesis environment then requires a large, verified library of design factors, ranging from code components to algorithm design theories, datatypes and their refinements, optimizations, architecture theories/patterns. It also requires an extensive library of theories about application domains. Construction of these libraries is capital intensive, but the investment will tend to be amortized over many applications. The amortization is particularly clear for

software families and product lines. Design patterns provide clear examples of design factors whose development of cost is amortized over a variety of (unrelated) applications.

We see the following basic features of synthetic approaches:

- Requirements modeling (specification) languages – These model the required properties of a system, including constraints on functionality, safety/liveness, performance, security, resource consumption in the host environment, fault tolerance, stability, and so on. Composition of specifications allows reuse and lower cost of evolution.
- Modular design factors – Separation of concerns suggests that design information be factored out into modules. Furthermore, developers want access to best-practice design knowledge (whether embodied in code-level components or in abstracted design patterns or theories). Design factors represent reusable theories about various classes of systems (via software architectures), algorithms, data structures, optimization rules, domain-specific design knowledge, and other kinds of design information that cross-cut system structure.
- Composition – Two forms of composition are needed. One composes specifications, allowing modular construction at the requirements level. The other composes a design factor with the developing design.
- Refinement – Refinement is the staged/incremental introduction of design information and the incremental analysis of properties as necessary; i.e. some properties can be preserved under refinement, others must be incrementally checked at the point that design information is introduced that could violate them. A developing artifact can be refined by composing it with a design factor.
- Automated support tools – for composition, refinement, and inference/analysis, as well as code generation.

One argument for the necessity of a synthetic approach comes from the phenomenon of dependencies in code. There is evidence that a key reason for the complexity and inertia of large-scale systems is that the number of dependencies tends to grow superlinearly with code size. It becomes very difficult to keep a system consistent and understand the impact of a change, when each change in an application can potentially have an impact in many places in the software.

This suggests that a design paradigm that treats software as a composition of parts that are relatively free of inter-dependencies, and systematically introduces dependencies in a manageable way, will serve to improve productivity. Two of our most important techniques for thus reducing dependencies are *modularity* (information hiding) and *abstraction*. Conversely, system optimization (information spreading) and refinement/specialization tend to increase dependencies. Hence, we are led to the paradigm of automating the composition and refinement of modules at as abstract a level as possible, with a consequent focus on requirements and design factors.

Modules at several levels of abstraction will play a role in synthesis. Legacy code is end result of quantities of design knowledge, but typically the knowledge is lost, and the many dependencies within the code make it difficult to understand, reason about, and adapt. Aspect-oriented

programming gets at new possibilities for increased modularity (at the programming language-level) with the concept of aspects and aspect composition (weaving). Generic programming similarly explores novel ways to increase programming language-level modularization by developing ways to make programs parametric on complex structures such as type constructors, algebras, O-O classes, and so on.

Examples of more abstract modules of design knowledge include intentions (as in Intentional Programming), classical program transformations, the design theories developed at Kestrel, and others.

To sum up, system development productivity can be increased by automated composition and refinement of requirement specifications and software design knowledge. The phenomenon of extensive dependencies in code and the difficulty of composition in the presence of dependencies moves us to the automated composition of design factors and requirements with minimal dependencies, i.e. at relatively abstract levels. Automation helps to systematically introduce and manage dependencies during refinement to code. Modularity of the composition helps to isolate change during evolution, lowering lifecycle costs and improving productivity.

2 A Synthesis Approach based on Category Theory and Logic

Kestrel's approach to automated synthesis allows capturing and classifying not only software components, but also higher-order process components, design knowledge and reusable patterns, in such a way that their routine application can be *automated*. The automated refinement steps preserve the specified process properties, and thus increase assurance, while decreasing the evolution costs. The approach is based on a full mathematical elaboration of the semantic foundations, allowing its implementation in a categorically based tool suite (the Specware, Designware, Planware, and EPOXI systems).

- Specification language – Specware is based on the category of higher-order logical specifications and their morphisms (interpretations). Sort constructors such as product, coproduct, subsort, and quotient are included in support of datatype refinement. On this foundation Epoxi models stateful behavior with abstract states and transitions corresponding to specifications and interpretations respectively.
- Modular design factors – Designware emphasizes taxonomies of design theories that embody precise machine-applicable knowledge about various classes of algorithms (e.g. divide-and-conquer, global search), data structures (e.g. sets implemented as red-black trees or bit-vectors, or hash tables), optimization rules (e.g. context-dependent simplification, finite differencing), as well as domain-specific design knowledge (e.g. a taxonomy of resource theories in the scheduling domain).
- Composition – Composition of specifications is carried out automatically by colimits in near-linear time. Since design theories are also captured as specifications, the colimit operation is also used to compose specs and design theories. The colimit operation naturally preserves properties expressed as axioms.

- Refinement – Automated support for two kinds of refinement are provided: (1) colimits are used to compose specs and design theories, and (2) metaprograms, called refinement generators, apply design knowledge to a specification resulting in a refinement of it. Typically the latter embody optimization transformations.
- Automated support tools – The colimit operation is computable in near-linear time, and scales well. Specware also connects to several theorem-provers for inference/analysis support.

3 Issues

- Automation – developing a mechanizable framework for software development and evolution; automation to increase productivity in the presence of scale and complexity.
- Evolution – achieving a better understanding of the representations and algorithms for propagating change through design structures.
- Process models – developing synthesis-oriented development models. Various roles include developers of design factors and domain theories, developers of system composition, refinement, and analysis tools, as well as application-oriented developers. Planware provides an example of how end-users can be shielded from the mathematical machinery underlying a synthesis development environment.
- Libraries of Specifications and Design Factors – building large libraries of specifications whose consistency has been established. Also, building large libraries of machine-processable design theories with verified properties (such as functionality, performance/timing, safety/liveness, fault tolerance, stability, etc.)
- Inference and analysis tools – combining general-purpose inference tools with decision procedures and other specialized inference and analysis tools. Flexible, efficient tools for witness-finding.

4 Promising Concepts/Foundations

- *Coalgebras* are useful for specifying (the properties and structure of) discrete and continuous behaviors, allowing the modeling of both discrete and hybrid systems and design knowledge about them. Composition and refinement of hybrid models also have a category-theoretic foundation (via colimits and interpretations).
- *Dynamic game theory* provides the foundation for open system design, composition and analysis of protocols
- *Design theories/patterns/aspects* have in common the modular capture of design knowledge, providing access to best-practice design knowledge, as well as automated access and application during system design.
- *Combined inference procedures* allow general-purpose reasoning together with decision procedures for specialized logics and inference problems. Improved witness-finding techniques are needed to support refinement processes..