Models, Patterns, and Generators for Embedded Systems

Gabor Karsai[1]
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

Model-based development of embedded information systems necessitates the use of generative techniques. Models determine and configure computations, but they also lend themselves to design-time analysis. This paper discusses why generative techniques are important for embedded systems, outlines how a general-purpose generator technology can be built to help the engineering of complex applications. However, a number of technical problems have to be solved before generators will become an everyday tool in embedded software development, related to the construction, verification, and usability of generators.

Introduction

The development of embedded systems is challenging because of the tight coupling between an information- processing component and the environment. The environment consists of a number of "processes": physical, cognitive, economical, societal, etc., which are in continuous interaction with the computational processes, resulting in a very complex dynamics. Obviously, low-level and often obscure software design decisions have a profound impact on the behavior of the software in this context, potentially leading to undesired overall behavior. It is fair to say, that current software design techniques do not address these interactions properly, and thus can lead to systems that fail disastrously. In short, the science and engineering of Embedded Information Systems (EIS) needs to be developed.

To give an example, let us consider the recent accidents of the V-22 tilt-rotor aircraft [1]. The accident investigation has concluded that the cause of the accident was two-fold: (1) a physical failure in a hydraulic line, and (2) a bug in the flight control software that "allowed huge torque and RPM changes when a reset signal was generated in conjunction with the hydraulic line failure condition". It is also worth mentioning that the standard operating procedure proscribed a reset of the flight control computer under fault conditions. The accident highlights complex interactions among a number of processes: the failure in a hydraulic line impacted the behavior of the software because the operational process also required the pilot to reset the computer.

There are complex interactions between the dynamics of the software and its physical environment as well. The laws of physics and mathematics (especially the sampling theorem) determine a dynamics that the external physical world expects from an EIS. On the other hand, software design decisions, like task allocation, scheduling policies, and word-length choices influence the behavior of the software, and thus its dynamics. The problem is how to resolve the differences between the expected and the real dynamics of the embedded system, and how the software design decisions have to be changed to comply with the expected behavior.

As current software engineering techniques and tools do not adequately address these needs, new solutions are needed [12]. The requirements of embedded systems introduce new complexities, and developers need sound engineering principles, techniques and tools for managing these complexities. In this paper, we show how a technique based on software generators: tools that generate executable code from high-level, domain-specific models can provide support in satisfying these requirements. As an additional benefit, generators can also support the use of design patterns, as it is discussed below.

Background

Although the dream of component-based software is more than 40 years old, it has not been totally fulfilled, and the lack of progress is especially apparent in embedded software [2]. On the other hand, component-based hardware is a reality since modern manufacturing techniques have been applied in industry. One possible reason is that while hardware components are very rigid and hard to change, software components are "fluid", and modifiable (either through an API at run-time or in source code at design time). This flexibility may be harmful, and work against large-scale reuse in practice. Another reason might be the complexity of the behavior and the interface of software components. Any hardware component (e.g. a CPU) has a well-defined

[1] gabor@vuse.vanderbilt.edu

interface, which it is "guaranteed" to comply with, and the designers and manufacturers very carefully monitor their production processes to ensure that no product is created which violates the constraints of the interface. Software interfaces are much more complex and much less well-defined than hardware interfaces. This is especially true regarding the timing behavior of components. For instance, very rarely has a software component a documentation that says: "this component, given any data as input, will generate a correct output within 100 microseconds". A third reason might be that it is not obvious how to calculate emergent properties of ensembles of software components. For hardware components the laws of physics applied by electrical engineering tell us, with a high degree of confidence, how a circuit will behave in a frequency domain. Given a set of software components, it is a very difficult problem to calculate the emergent timing properties of the ensemble from the properties of the components.

All of the above three problems: component fluidity, behavioral and interface complexity, and the lack of component calculus make the composition of embedded software systems an extremely complex process. The situation is degraded further by the fact that most of the composition happens through editing source code, where it is very hard to keep track of what is exactly happening. Although recent advances in using visual tools for software design and synthesis indicate some progress in the right direction there is still plenty of room for improvement.

The generative approach

One technology that shows great promise in solving the composition problem for embedded information systems is the use of *software generators*. Generators [3] are tools that synthesize source code (or its equivalent, for instance an augmented syntax tree that can be the input of a machine code generator) from some "higher-level" input. The key difference between generators and language compilers is that generators operate on domain-specific, possibly ad-hoc defined input, while compilers operate on source code with well-defined syntax and semantics.

We call the input of the generator *model* to emphasize the difference between it and source code in a programming language. A model captures some domain-specific and relevant information that directly determines and influences the output of the generation. The model can be viewed as abstract description of the architecture of the embedded software, although it may also include mathematical models of the physical component of the embedded system, and the physical environment. It should be noted however, that models may include source code in some programming language, that is "passed through" by the generator to the final compiler.

We already do see the impact of generators on embedded systems. Matlab [4], Matrix-X [5], "Software through Pictures" [6], and LabView [7] are just a few examples for generating code from high-level models, without having to deal with the problem of software componentization mentioned above. Many embedded applications have been successfully developed using these tools. However, these tools achieve their results by using limiting component interfaces, and the methods of composition by predefining some supported "models of computation" [2], and do not support calculation of properties of ensembles of components. On the other hand, they do show that the generator-based approach to embedded software composition is a viable technology.

Generators are inherently necessary for embedded software development, if we want to step beyond the level of complexity manageable by current processes. Current processes are characterized by the use of design techniques (e.g. SDL or UML) not always adequate for describing highly reactive systems, middle-level programming languages (e.g. C or C++), and debugging on the level of execution or below it (e.g. by using logic analyzers). Programming languages (especially new, very high-level, domain-oriented languages) do offer some help, but alone they are unable to address all the needs of software composition. The main reason for it is that developers invent (and use) novel ways of component composition, and if the language did not anticipate that style of composition, then it becomes very awkward to use. What is needed is some "programmable compiler", which allows the developer to express a composition style that becomes a "first-class" concept of the language. Arguably, generators, and especially user-extensible generators can support this activity.

To summarize, embedded software composition needs tool support to manage complexity arising during the development, deployment and maintenance of the systems, and software generators seem to fill an important role in these processes.

Generators

Generators of embedded software can solve the three problems mentioned above as follows.

1. *Component fluidity*: The generator can take the source code of a fluid component and morph it to the needs of a particular application, without human intervention. A component can be coded in a highly parameterized way, and if it is composed of other components, the composition specifications may contain conditional expressions. A conditional expression encodes a design rule that instructs the generator to emit different code depending on higher-level requirements. For example, a conditional expression may choose between two implementations of a lookup-table depending on the number of expected to be looked up:

   ```
   if sizeof(table) < 8 then use(Array) else use(Hashtable)
   ```

   Generators can determine, at composition time, the parameter values for the components and adjust them accordingly.

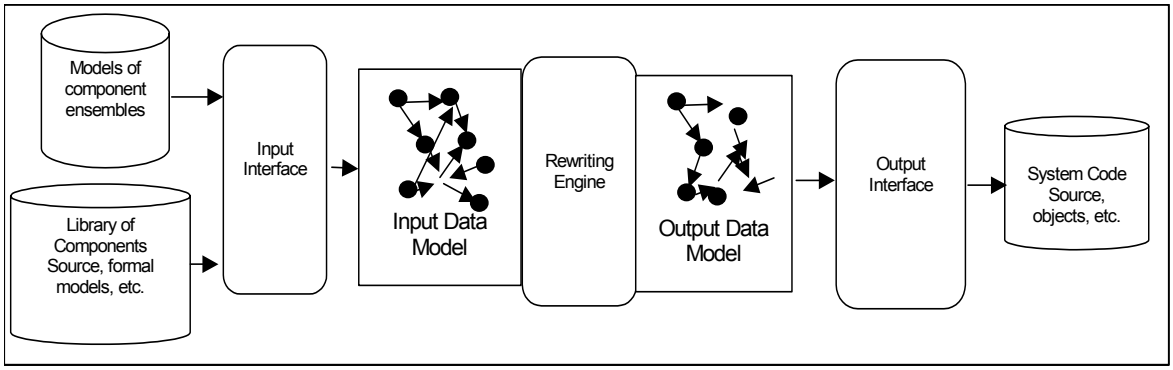
2. *Component integration:* Provided a component is properly described and a formal and symbolic model (of desired quality) of the component is available, the generator can detect mismatches between component interfaces and behavior, and either signal an error for the developer or generate "glue-code" automatically that "fits together" the components. This automatic insertion of type-converters between components has been done in the past for simple data types, but it can possibly be extended to behaviors as well. Of course, automatic type conversion may introduce undesired situations (like possible constraint violations on data), but a generator should be able to detect this and give a warning to the developer to prepare for this contingency.
3. *Ensemble properties*: A generator can be written such that it has formal "knowledge" of the properties of the components it is integrating. For instance, a generator may have a formal description of the timing properties of a component. If formal rules for calculations are available, then the generator can calculate the emergent properties of the component assemblies. For instance, if the worst-case run-time of a component is known, then a generator can calculate the worst-case timing of a component ensemble provided all the component communication patterns are known. This latter can be easily enforced by using a simple composition and scheduling technique (e.g. synchronous dataflow network). While the "general calculus" for component ensemble properties has not fully been developed yet, for some specific cases there are results available (e.g. RMA [8]).

Software generators for embedded systems have to exhibit some degree of flexibility. The rationale here is that sophisticated designers tend to invent their own abstractions, and wish to translate those into efficient code. If the abstraction is realized only as a certain coding pattern, it is almost impossible to reuse it across different projects. A generator-based approach can help in the sense that the designer can define a new domain-specific language (DSL) construct for the new abstraction, and determine its interpretation by writing, modifying, or extending a generator such that it "understands" the new construct. Thus, we envision that skilled designers extending the generators they use.

Generator architecture

A generator is similar to a compiler, although simpler: it rarely has to deal with optimizing executable code, for instance. In the most generic sense, a software generator has three parts: an input interface, a rewriting engine, and an output interface. These parts form a pipeline as shown on the figure below.

Similar to the pipeline structure of traditional compilers, the input interface reads the models: formal descriptions, source code or some other representation of components and component assemblies, and builds in internal data structure that is the input to the rewriting engine. The input interface corresponds to the lexical and syntactical analyzer parts of a compiler. However, unlike in compilers, the generators often use graph-like data-structures directly (as opposed to text-based input). For example, in the case of Matlab, the "input" the generator is a Simulink diagram consisting of a network of blocks and connections), and not textual source code.



The rewriting engine builds an "output" data structure from which the output interface generates source or object code, or system configuration files. This output data-structure is then traversed and "printed" in some form. The output of the generator may be source code (e.g. C++ or Java), system configuration files, or binary data. A key point of the generative approach is that *all* these outputs are derived from the same, single input, thus its consistency is automatically enforced. If a change is necessary, one changes the input of the generator (never the output!), and re-runs the generator.

Naturally, the key to the generators is the rewriting engine. This component performs a graph transformation on the input data: the input model, with possibly calculating and validating properties of the ensembles being synthesized. There are at least two approaches to implementing this rewriting engine: (1) Define the rewriting actions in the form of "input pattern -> output pattern" mappings, and use a generic traversal strategy (e.g. "apply patterns exhaustively") to perform the rewriting. This approach is very easy for the user who wants to specify rewriting rules, but computationally may not be efficient. (2) Define a specific traversal strategy for the rewriting such that when visiting a node a portion of the output data model is created and connected to other

parts of the output. This approach requires lower-level coding than the previous one, but the user has full control over traversal strategy, and thus it is computationally more efficient. Both of these approaches can be implemented using straightforward procedural code. However, for convenience, the supplier of the generator software technology may provide a small, domain-specific language to program the translator component. This DSL is specifically tailored for the domain of generators and it can support the rapid prototyping, and modification of the rewriting component.

The patterns connection

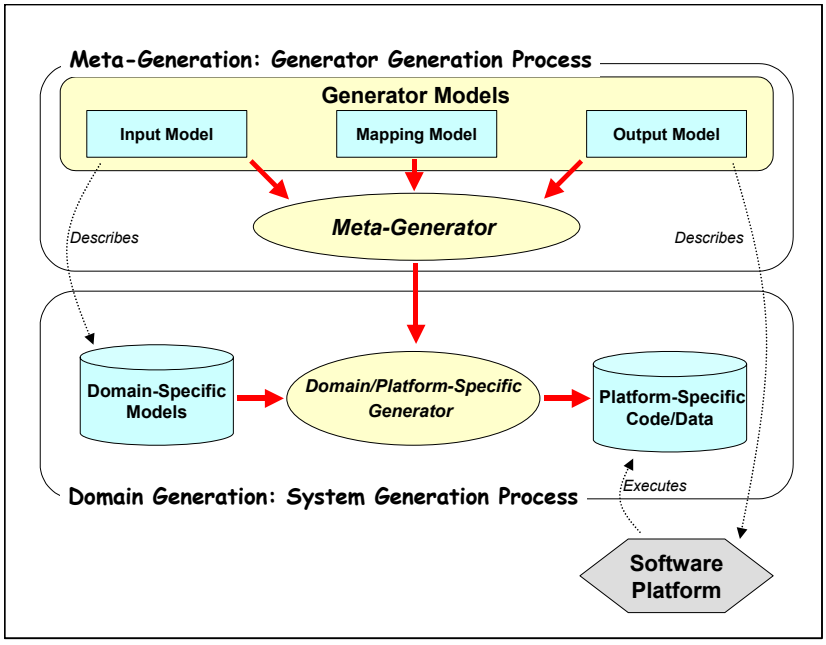
The Design Patterns community has identified a number of useful prototypical solutions to recurring software design problems. Since the arrival of the seminal book [9] on the design patterns, serious work has been done on codifying and documenting the patterns encountered in software designs [13]. However, it is somewhat surprising that, compared to the overall impact and relevance of patterns, there is very little tool support for patterns. While some patterns can be implemented using, for instance, the template mechanisms in C++ [3], there are no "languages" for specifying patterns.

We believe generators can help supporting a more formal approach to pattern-based software development. A number of design patterns, especially the ones related to implementation, can be expressed as a collection of "class fragments", which cooperatively solve a design problem. Generator techniques can possibly be used to "weave" these patterns into "base-level" code, not unlike aspect-oriented programming techniques can weave aspect-code into "base-level" code at specific points [14]. We believe generators can easily perform source-to-source transformation as well, and verify the validity of the pattern composition during the process.

The tools

We are working on a set of tools for building generators for embedded systems [13] that can efficiently transform components and their models, and models of component ensembles into code for running systems. These tools will allow the easy specification and customization of generators by sophisticated end-users, who want to create and possibly reuse their own generators, or any portion of those.

The tools are using a generator-based approach as well: the embedded software is generated from a formal model of the input and output data models, and the mapping between the two. The figure below shows a notional architecture for generating and using a generator.



We call the top, first stage as "meta-generation": this is done when the generator is built (or re-built). The second stage at the bottom takes place when the generator is actually used by the embedded system designer, and its task is to translate the domain-specific models into platform-specific code. Obviously, the meta-generation is a rather infrequent process, while the system generation is performed rather frequently.

We use a technique called *metamodeling* for specifying components of the generators. A meta-model is a description of a modeling language, like a context-free grammar in Backus-Naur Form is a description of a textual language. The difference is that the meta-model can describe not only syntax for linear text, but also (1) all the legal combinations of objects (i.e. an object

"graph"), and (2) other, non-structural constraints (e.g. "the sum of these three attributes of an object should be equal to a constant"). We use the industry-standard [11] language of UML class diagrams and constraints, OCL, as our metamodeling language. These meta-models are then used to describe the input and output data models of the generator. These descriptions are then used by our tools to synthesize C++ object definitions that allow easy access to the "input models" and the output data structures. This techniques allows us to couple the generators to arbitrary model repositories, for instance, engineering databases, modeling tools, or XML files.

The mapping used on the generator can involve arbitrarily complex computations to support flexible composition. The mapping model will be expressed in another domain-specific language tailored for expressing the main work of the generator: the rewriting of the input data-structures into the output. The mapping model language will provide support for describing automatic or user-specified traversal strategies, with the possibility of using constraints to guide the traversal or perform multiple passes over the input data-structures. In our experience, these techniques make the writing and modification of translators a very rapid process. This approach will also allow incremental operation such that a small change on the input will result a small change in the output, which is important for interactive development. We will also attempt to create generators whose footprint and computational requirements makes them suitable for embedded deployment, on the run-time system itself. This novel technique offers new capabilities for embedded software where a running system can regenerate and modify its own component architecture at run-time. The first results and the documentation of the project are available on our website [13].

Summary, conclusions, and future work

Complex embedded information systems require sophisticated yet productive development practices. We argue that domain-specific modeling and languages, coupled with automatic software generation technology can help supporting this process. The domain-specific models allow the system designers to focus on domain-specific problems, while the software engineering aspects of the work are addressed by the generators itself. We have presented a generic architecture for software generators, and discussed our techniques for the synthesis of generators. The use of automatic synthesis in building a non-trivial, yet crucial piece of software: the generator will allow sophisticated designers to extend its capabilities and introduce new abstractions in the system.

We strongly argue that embedded system development and system integration cannot be managed without the use of automated generation techniques. Even now, major manufacturers who build large-scale embedded systems do use generation technology, although it is often based on simple text manipulation. Further research and development is needed to define new modeling paradigms and languages for embedded systems, real-time software components and composition techniques that support the designers. Furthermore, research should target how generators can be made extensible, and how complex analysis techniques can be coupled with generators to determine relevant properties of the generated system.

Acknowledgement

The DARPA/ITO MOBIES program (F30602-00-1-0580) is supporting, in part, the activities described in this paper.

References

1. http://www.defenselink.mil/news/Apr2001/t04052001_t405mv22.html
2. Edward Lee: "What's Ahead for Embedded Software?", IEEE Computer, pp.18-26, September,2000.
3. Czarnecki, K. Eisenecker, U: *Generative Programming - Methods, Tools, and Applications,* Addison-Wesley, 2000.
4. http://www.mathworks.com/
5. http://www.isi.com/
6. http://www.aonix.com/
7. http://www.ni.com/
8. Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza: *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Pub; 1993.
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*, Addison-Wesley, 1995.
10. http://hillside.net/patterns
11. http://www.rational.com
12. Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," *IEEE Computer*, pp. 110-112, April, 1997.
13. http://www.isis.vanderbilt.edu/Projects/mobies/default.html
14. http://www.aosd.net/