IEEE Instrumentation and Measurement
Technology Conference
Budapest, Hungary, May 21-23, 2001.

On Generators for Embedded Information Systems

Gabor Karsai, Akos Ledeczi, Miklos Maroti
Institute for Software Integrated Systems, Vanderbilt University*
Nashville, TN 37235, USA
Phone: +1 615 343-7460, Fax: +1 615 343-7440
Email: gabor@isis.vanderbilt.edu, URL: http://www.isis.vanderbilt.edu

Abstract *– The sophisticated services provided by modern measurement systems and complex instruments are implemented almost exclusively in software. In fact, these instruments are Embedded Information Systems where software components implement complex functions and act as the system integrator. The development of these software systems is crucial for achieving the desired quality and precision in a measurement system. In this paper, we present an approach to the development of complex embedded software systems through the use of generators. Software generators are software engineering tools that translate high-level, domain-specific models into executable systems. The key elements of this technology are domain modeling and automatic code generation resulting in the ability to reuse design solutions.*

Keywords *– embedded information systems, embedded software, software generators*

I. INTRODUCTION

The advanced capabilities of today's measurement systems are, to a large degree, due to information technology (IT). However, the IT used in these systems is radically different from the everyday IT of office software and the world wide web: it is the IT of embedded information systems. In that respect, it bears closer relationship to avionics software suites of aircraft and real-time process control applications than word processing. Similar to those embedded systems, the expected quality and reliability of a measurement system is of the utmost importance, and it may even be classified as "high-consequence" in certain applications.

Embedded Information Systems (EIS) require rather sophisticated software designs, as the software is in a tight coupling with its physical environment. The software of a measurement system running on a particular hardware device controls all the measurement processes, manages hardware resources, and is responsible for the user interface, in addition to being forced to respond to real-time events under strict timing constraints.

There are also complex interactions between the behavior of the software and its physical environment. The laws of physics and mathematics (especially the sampling theorem) determine a dynamics what 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 necessary. The requirements of embedded systems introduce new complexities; developers need sound engineering principles, techniques and tools for managing them. In this paper, we show how our technology based on software generators – tools that generate executable code from high-level, domain-specific models – can support satisfying these requirements.

II. BACKGROUND

Although the dream of component-based software is more than 40 years old, it has not been totally fulfilled. 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 for a long time. 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). The disadvantage of this tremendous flexibility is that it works 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 interface, which it is "guaranteed" to comply with. The designers and manufacturers monitor their production processes very carefully to ensure that no product is created that 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 does a software component documentation state that "this component, given any data as input, will generate a correct output within 100 microseconds."

A third reason might be difficulty to calculate emergent properties of ensembles of software components. For hardware components, the laws of physics applied by

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

0-7803-6646-8/01/$10.00 ©2001 IEEE

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 these problems, component fluidity, behavioral and interface complexity, and the lack of component calculus, make the composition of embedded software systems extremely complex. The situation is exacerbated further by the fact that most of the composition happens through the manual editing of source code, where it is very hard to keep track of component interactions. Although recent advances in visual tools for software design and synthesis indicate some progress in the right direction, there is still plenty of room for improvement.

III. THE GENERATIVE APPROACH

One technology that shows great promise in solving the composition problem for embedded information systems is the use of software generators [8]. Generators are tools that synthesize source code (or its equivalent, for instance an augmented syntax tree that can be fed to a machine code generator) from some "high-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 a *model* to emphasize the difference between it and source code in a programming language. A model captures some relevant domain-specific 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 components of the embedded system (e.g. the measurement instrument), 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 [3], Matrix-X [4], "Software through Pictures" [5], 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, they achieve their results by using limiting component interfaces and methods of composition by predefining a small set of fixed "models of computation" [2]. They 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 necessary for embedded software development in order to be able to step beyond the level of complexity manageable by current processes. The current state-of-the-art is 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 they are unable to address all the needs of software composition alone. The main reason for it is that developers invent (and use) novel ways of component composition. If the given language did not anticipate that style of composition, then it usually becomes awkward to use. What is needed is some "programmable compiler" allowing 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 the complexity arising during the development, deployment and maintenance of the systems. Software generators can fill an important role in these processes.

IV. 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 match the needs of a particular application without human intervention. A component can be coded in a highly parameterized way. 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 items expected to be looked up:

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

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

2. Component integration: Provided a component is properly described and a formal and symbolic model of it (of desired quality) is available, the generator can detect mismatches between component interfaces and behavior and either signal an error for the developer or generate "glue-code" that matches the components automatically. This automatic inser-

tion of type-converters between components has been done for simple data types in the past, but it can be extended to behaviors as well. Of course, automatic type conversion may introduce undesirable side effects (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 also 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 also known. This latter condition can be easily enforced by using a simple composition and scheduling technique (e.g. synchronous dataflow). While the "general calculus" for component ensemble properties has not been fully developed yet, there are results available for some specific cases (e.g. RMA [9]).

Software generators for embedded systems have to exhibit some degree of flexibility. The rationale for this 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 augment the domain-specific language (DSL) with a new 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 skilled designers extending the generators they use.

V. 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 in Figure 2.

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 an 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 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 to 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 that 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 multiple outputs are derived from the same, single input, thus their consistency is automatically enforced. If a change is necessary, one needs to modify the input of the generator (never the output!), and re-run it.

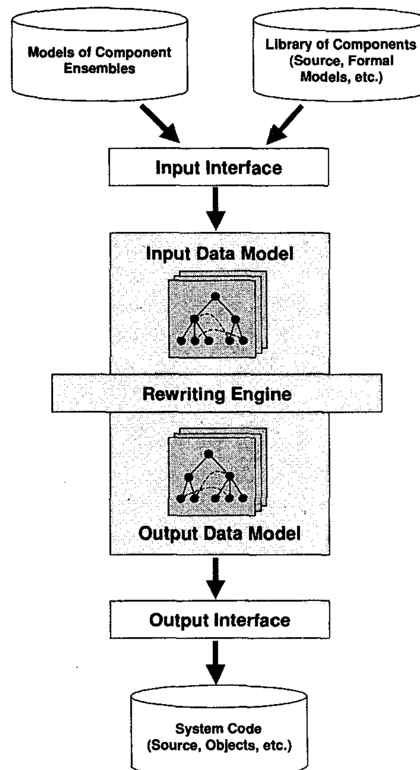

Figure 1. Generator Architecture

Naturally, the key to the generator is the rewriting engine. This component performs a graph transformation on the input data: the input model. It may involve 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 to 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 out-

put 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, thus, it is computationally more efficient.

We are working on an integration of the transformational and operational approaches. By integration we mean that the mapping is specified as an explicitly controlled transformation process interleaved with actions. Instead of giving generic rules for the transformation, we want to precisely control when and how the transformations are to be applied. If necessary, a transformation can result in a "side effect" (i.e. an action in terms of the output platform), but these actions are explicitly marked as such. Note that the approach strikes a balance between the fully formal representation of the mapping and the highly practical (but much less formal) operational view. Our goal is to develop a method that is formal enough for analysis while practical enough for everyday use.

The salient properties of the approach can be described as follows.

1. The traversal paths of the input are explicitly specified. The traversal path (or sequence) describes the order in which the nodes of the input graph should be visited. This description can be done using a method employed in Adaptive Programming [11], in terms of node types and the "edges" (i.e. class relations) the traversal should follow.

2. The transformational steps are explicitly specified and tied to the traversal specifications. A transformation specifies how a portion of the input graph (a "context") is to be transformed into a portion of the output graph. We will develop a method for capturing the input context and for describing the construction of the output data structure.

3. The transformation may result not only in a data structure, but also in an action on the software platform. These actions should be selected from the actions defined in the output model. Because the output model includes pre- and post-conditions for actions, these actions lend themselves to static and dynamic verification. Static verification means that traversal sequences are generated and the pre- and post-conditions are verified for each action sequence. This static analysis can possibly be performed using symbolic state-space exploration techniques (e.g. model checking). Dynamic verification means that pre- and post-conditions can be monitored and checked during run-time. In case of a failure, an error handling operation can be triggered.

This approach can be implemented using straightforward procedural code. However, for convenience, a simple, domain-specific language to program the translator component can be provided. This DSL is specifically tailored for the domain of generators and it can support the rapid prototyping, and modification of the rewriting component.

VI. THE TOOLS

We are working on a set of tools for building generators for embedded systems [6] 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 them.

The tools are using a generator-based approach as well: the generator component itself 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.

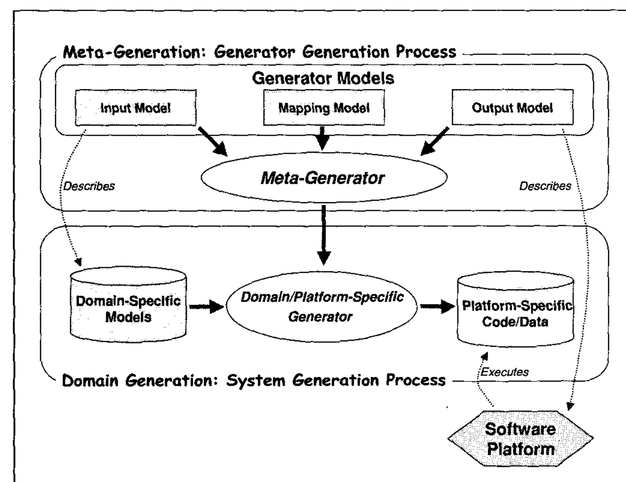

Figure 2. Generator generation

We call the top, first stage the *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 more often.

We use a technique called metamodeling for specifying components of the generators. A meta-model is a description of a modeling language, like the Backus-Naur Form is a description of a textual language. The difference is that the meta-model can describe not only syntax, but also constraints: (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 UML class diagrams and OCL constraints [10] as our metamodeling language. These meta-models are 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, an important consideration for interactive development. We will also attempt to create generators whose footprint and computational requirements make 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 [6].

VII. THE MEASUREMENT SYSTEM CONNECTION

Modern measurement systems often require complex software packages for implementing sophisticated services. The software adds significant value to a system, and its quality is crucial to the usability of the entire application. A significant effort is required in developing this software, which is highly specialized code for a highly specialized embedded system. We argue that domain-specific languages will have a significant impact on developing complex measurement systems. The reason is that instrument designers are experts in instrument design, not necessarily in software design, and can be much more productive using a domain-specific language (DSL) than a procedural programming language. Designing high-quality embedded information systems is a complex and error-prone process. The use of DSL-s and software generators will help in putting the knowledge of the skilled embedded system designers into an automated framework, and make that knowledge available to measurement specialists.

VIII. CONCLUSIONS AND FUTURE WORK

The development of complex embedded information systems used in complex measurement systems require sophisticated and productive development practices. We argue that domain-specific modeling coupled with automatic software generation technology can support this process. The domain-specific models allow the system designers to focus on metrological problems, while the software engineering aspects of the work are addressed by the generators themselves. 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 believe that embedded system development and system integration cannot be managed without the use of automated generation techniques. Even now, major manufacturers building 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.

REFERENCES

[1] Janos Sztipanovits and Gabor Karsai, "Model-Integrated Computing," IEEE Computer, pp. 110-112, April, 1997.

[2] Edward Lee: "What's Ahead for Embedded Software?," IEEE Computer, pp.18-26, September, 2000.

[3] http://www.mathworks.com/

[4] http://www.isi.com/

[5] http://www.aonix.com/

[6] http://www.isis.vanderbilt.edu/Projects/mobies/default.html

[7] http://www.ni.com/

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

[9] 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.

[10] http://www.rational.com

[11] Lieberherr,K.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, Boston, MA, PWS Publishing Company, 1996