LNCS 2487: Proceedings of Generative Programming and Component Engineering, pp 32-49, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA October 2002

Generative Programming for Embedded Systems

Janos Sztipanovits and Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University, P.O. Box 1829 Sta. B. Nashville, TN 37235, USA
{sztipaj,gabor}@vuse.vanderbilt.edu

Abstract. Embedded systems represent fundamentally new challenges for software design, which render conventional approaches to software composition ineffective. Starting with the unique challenges of building embedded systems, this paper discusses key issues of model-based technology for embedded systems. The discussion uses Model-Integrated Computing (MIC) as an example for model-based software development. In MIC, domain-specific, multiple view models are used in all phases of the development process. Models explicitly represent the embedded software and the environment it operates in, and capture the requirements of the application, simultaneously. Models are *descriptive*, in the sense that they allow the formal analysis, verification and validation of the embedded system at design time. Models are also *generative*, in the sense that they carry enough information for automatically generating embedded systems from them using the techniques of program generators.

1 Introduction

Perhaps the biggest impact of the "IT explosion" in the last decade has been the emerging role of computing and software as "universal system integrator." Systems are formed from interacting components. The new trend is that an increasing number of components and interactions in real-life systems, which were previously physical, are becoming *computational*. From large-scale systems, such as manufacturing processes or command and control (C^2) systems, to small systems, such as automobiles and simple appliances, interaction and coordination of physical components increasingly involves digital information processing and communication. For example, the automotive industry is currently testing "brake-by-wire" systems, where the currently dominant hydraulic and mechanical brake systems will be replaced by position sensors observing the brake pedal, electrical actuators exerting braking force, and a distributed, embedded computer system, which computes the optimum force distribution according to the driving conditions. Two important consequences of this change are the following:

- **Increased fusion of software into application domains**. In many application domains, computing has become the focal point of complexity and the primary source of new functionality. For example, over 90% of innovations in the automotive industry come from embedded computing [33]. The increased

significance of computing means that unless unique characteristics of the application domain are reflected directly in the software development paradigms, application engineering considerations must be mapped manually onto general-purpose software engineering concepts and tools, which is tedious and error-prone for both the domain experts and the software experts. The difficulty of this manual mapping process creates the need for sharply tailored capabilities, such as domain-specific languages, generators, and composition platforms that enable building of applications from components.

- **Physicality in software design.** In embedded computing applications, the role of the embedded software is to configure and control the operation of programmable computing devices so as to meet physical requirements at their sensor-actuator interfaces This deep integration of computing with physical systems implies that essential physical characteristics of systems (such as latency, noise, power consumption) are strongly influenced — or simply determined by — software. Consequently, software requirements become multi-faceted, i.e., computational platforms and software must satisfy functional and physical requirements *simultaneously*.

The goal of this paper is to discuss challenges and opportunities of generative programming for embedded software development. We use the term "generative programming" in a broad sense: systems or components of systems are produced automatically from abstract terms [28]. The examples for possible solutions are based on our experience gained from the *Model-Integrated Computing (MIC)* effort at the Institute of Software Integrated Systems (ISIS) at Vanderbilt University [1].

The outline of the paper is the following: In Section 2, we examine the challenges of composing complex systems from components and describe a model-based extension of composition platforms. Section 3 summarizes the challenges and discusses the MIC approach in modeling languages and model building. In Section 4, we provide an overview of generator technologies in the MIC framework. Section 5 summarizes some of the relevant approaches and compares them with MIC.

2 Significance of Generative Programming

Composition and component-based design are key tools in modern software engineering for managing complexity. The concept of component-based design is straightforward: systems are built by composing software components with precisely defined interfaces using standardized interconnection mechanisms. "Plug-and-play" construction is supported by an underlying composition framework, such as CORBA or COM+, which facilitates the component interactions by providing standard services such as a request broker, interface repository, location service, and others. Unfortunately, this solution tends to work either for building systems with relatively coarse-grained components using small amounts of "glue code", or for very small systems with a few components because of the following two problems:

1. Component interfaces in conventional standards-based composition frameworks only capture the signature, but not the semantics of components. Consequently, design integrity for the composed system may quickly be lost and inconsistencies

may emerge by simply combining components based on compatibility of interface signatures.

2. To achieve flexibility, components are frequently designed to be customizable to different application contexts via parameterization or via the selection of alternative implementations. In large systems, these choices represent complex, interacting assumptions about operating conditions, which may lead to brittleness.

The success of building applications based on coarse-grained components is the result of strong restrictions on component interactions: the dominant components (such as databases or web browsers) preserve the overall design integrity. In case of small systems, the system designers can preserve design integrity without extensive tool support through heroic manual effort.

Although software composition is rarely easy, the problems of software composition for embedded systems are particularly hard. Physical processes surround embedded computers, which receive their inputs from sensors and send their outputs to actuators. When viewed from their sensor and actuator interfaces, embedded computing devices act like physical processes with dynamics, noise, fault, size, power and other physical characteristics. *The role of the embedded software is to "configure" the computing device in order to meet its physical requirements* [2]. The mapping of logical behavior into physical behavior is influenced by the detailed physical characteristics of the devices involved, including their physical architecture, instruction execution speed, bus bandwidth, power dissipation, etc. In addition, modern processor architectures introduce complex interactions between the software and essential physical characteristics of the underlying devices.

It is not surprising that using current software technology logical/functional composability does not imply physical composability. In fact, *physical properties are not composable*, rather, they appear as cross-cutting constraints in the development process. The effects of such cross-cutting constraints can be devastating for the design. Meeting specifications in one part of the system may destroy performance in others, and, additionally, many of the problems will surface at system integration time rather than during unit development and testing. Consequently, we need to change our approach to the design of embedded software: productivity increases must come from tools that directly address the design of the whole system with its many different physical and logical aspects.

There are several comprehensive approaches, such as Model-Integrated Computing (MIC) [1], Aspect-Oriented Programming [3], Intentional Programming [4], GenVoca Architecture [5], that attempt to answer problems of composing complex systems. In our discussion, we focus on MIC and point out some of the similarities and differences with the other approaches.

Figure 1 shows an overview of the MIC architecture. The applications and infrastructure software (left side) are defined by application models and platform models. The difference between MIC for embedded and non-embedded systems is particularly significant regarding the scope and composition of models. In order to make the physical properties of the embedded system computable and analyzable, models capturing only the logical characteristics of applications and infrastructure software are insufficient. The models must include physical properties of the platforms and the mapping between the application and platform models. The scope of modeling and the required level of abstraction are highly domain-specific. We