Managing Embedded Systems Complexity with Aspect-Oriented Model-Driven Engineering

CORMAC DRIVER, SEAN REILLY, ÉAMONN LINEHAN, VINNY CAHILL, and SIOBHÁN CLARKE
Trinity College Dublin

Model-driven engineering addresses issues of platform heterogeneity and code quality through the use of high-level system models and subsequent automatic transformations. Adoption of the model-driven software engineering paradigm for embedded systems necessitates specification of appropriate models of often complex systems. Modern embedded systems are typically composed of multiple functional and nonfunctional concerns, with the nonfunctional concerns (e.g., timing and performance) typically affecting the design and implementation of the functional concerns. The presence of crosscutting concerns makes specification of adequate platform-independent models a significant challenge. Aspect-oriented software development is a separation of concerns technique that decomposes systems into distinct features with minimal overlap. In this article, we illustrate how Theme/UML, an aspect-oriented modeling approach, can be used to separate embedded systems concerns and reduce complexity in design. We also present Model-Driven Theme/UML, a toolset for model-driven engineering of embedded systems that supports modularised design with Theme/UML and automatic transformations to composed models and source code.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Coding Tools and Techniques

General Terms: Design, Theory

Additional Key Words and Phrases: Model-Driven Engineering, Aspect-Oriented Software Development, Theme/UML, Nonfunctional requirements, separation of concerns, model transformation, code generation

ACM Reference Format:

Driver, C., Reilly, S., Linehan, É., Cahill, V., and Clarke, S. 2010. Managing embedded systems complexity with aspect-oriented model-driven engineering. ACM Trans. Embedd. Comput. Syst. 10, 2, Article 21 (December 2010), 26 pages.
DOI = 10.1145/1880050.1880057 http://doi.acm.org/10.1145/1880050.1880057

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero-the Irish Software Engineering Research Centre (www.lero.ie).
Author's address: Distributed Systems Group, School of Computer Science and Statistics, Trinity College Dublin, Ireland; email: {*firstname.lastname*}@scss.tcd.ie.
Permission to make digital or hard copies part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1539-9087/2010/12-ART21 $10.00
DOI 10.1145/1880050.1880057 http://doi.acm.org/10.1145/1880050.1880057

1. INTRODUCTION

Model-driven design of embedded systems traditionally focuses on the use of verifiable formal models to describe system behavior at a high level of abstraction before decisions are made regarding the decomposition of the system into hardware and software. Automatic synthesis from high-level models is used to ensure that implementations are correct by construction [Edwards et al. 1997]. Outside the embedded systems world, model-based design has become a popular object-oriented software development paradigm, primarily due to the publication of the Object Management Group's Model-Driven Architecture (MDA) [Miller and Mukerji 2003]. Although MDA originally targeted large-scale enterprise applications [Oliver 2003], its underlying principles have been widely adopted and applied to the development of applications in a diverse range of domains, including embedded systems. This style of software development, commonly referred to as model-driven engineering (MDE) [Schmidt 2006] is complementary to the traditional embedded systems view of model-based design, focuses developer attention on the specification of generic solutions at the model level and facilitates subsequent progression to working systems through automatic model-to-model and model-to-text transformations. It is this notion of model-based design and its application to embedded systems software design and implementation that we focus on in this article.

The MDE principles of specifying abstract solutions and providing automatic transformations have the benefit of addressing issues surrounding platform heterogeneity and code quality. Platform heterogeneity, which typically necessitates multiple similar implementations of the same system, is addressed through abstraction of a generic solution to the problem at hand and automatic production of platform-specific versions of this solution. Improvements in system quality are achieved by means of automating the production of system code from models, thereby reducing the potential for introduction of human errors during system implementation.

In order to take advantage of model-driven engineering, it must be possible to design adequate system models. Recent advances in the capability and accessibility of embedded systems technologies have resulted in the capacity to build increasingly complex systems that consequently are difficult to model. Embedded systems are typically composed of many functional and nonfunctional concerns, with nonfunctional quality of service concerns (e.g., timing and performance) manifesting themselves as crosscutting concerns with system-wide impact [Deng et al. 2006; Wehrmeister et al. 2007]. Crosscutting concerns cannot be completely separated from the functional concerns that they affect and result in complex designs characterised by the presence of tangled modules. Tangled modules are those that represent multiple concerns and are, therefore, difficult to understand and change. Additionally, the need to repeat crosscutting code throughout a system results in the risk of inconsistencies at each point of use, further complicating development [Kiczales et al. 1997]. Failure to achieve adequate modularization negatively impacts key system properties such as extensibility, comprehensibility, scalability, reliability, and traceability [Filman et al. 2005].

Separation of concerns is an established technique for managing system complexity though modularization of distinct concerns [Dijkstra 1976]. Although traditional design and implementation paradigms provide some support for modularization, it has been established that total separation of concerns is not possible with approaches such as object orientation. Aspect-oriented software development extends the modularization capabilities of the object-oriented paradigm and has demonstrated effectiveness at separating crosscutting concerns in a wide range of domains [Greenwood et al. 2007; Laddad and Alexander 2003; Munnelly et al. 2007; Zhang and Jacobsen 2003], including the embedded systems domain [Afonso et al. 2007; Wehrmeister et al. 2007; Sousan et al. 2007].

Power management is one example of a critical embedded system design concern that is often poorly modularized in traditional design approaches [Selvaraj and Jozwiak 2005]. In battery-powered applications such as remote controls, mobile phones, and sensor networks, battery life can be preserved by switching to idle mode at times when no processing is required. This course of action results in power management source code, for example, `PCON |= 0x01;` in Intel 8051 systems, repeatedly appearing throughout functionally distinct operations. Theme/UML can reduce the presence of this type of crosscutting concern in embedded systems design.

Theme/UML is an aspect-oriented extension to UML that supports fine-grained decomposition and composition of both functional and nonfunctional concerns, including those that are crosscutting. Separated functional concerns are designed using standard UML to model each concern individually, with Theme/UML composition techniques addressing overlaps. From a nonfunctional concern perspective, Theme/UML's focus on complexity management through modularization captures those concerns that will manifest as code in the system. Using Glinz's taxonomy, nonfunctional concerns, in general, are categorized as attributes and constraints [Glinz 2007]. Attributes are separated into performance requirements and specific quality requirements. Performance requirements include quantitative measures such as timing, speed, volume, and throughput. Quality requirements cover the "ilities": reliability, usability, security, availability, and so on. Constraints capture physical (e.g., memory management), legal, cultural, environmental and interface requirements among others. Of these categories, the design of requirements that manifest themselves as behavior in the system, such as memory management (an example in this article's case study), can be simplified with Theme/UML's modularization techniques. In general, the code of such systemic, nonfunctional requirements cuts across the system as a whole, so their modularization significantly reduces their complexity. Theme/UML does not address the design of nonfunctional concerns that do not manifest as code in the system, but is complementary to existing techniques for capturing such requirements. Where the state of practice is to use UML (e.g., OCL constraints for some quantitative measures), Theme/UML is an extension on top of such techniques. See Section 3 and related work for more details on how Theme/UML and other approaches complement each other.

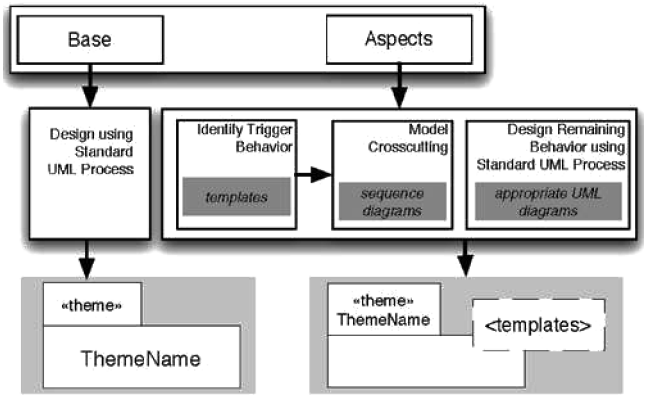

Fig. 1. Designing with Theme/UML.

In this article, we illustrate how Theme/UML can aid in reducing the complexity of embedded systems designs through its support for advanced modularization.[1] We present tool support for model-driven engineering of embedded systems based on Theme/UML models. The toolset, called Model-Driven Theme/UML, supports separation of concerns in design with Theme/UML, automatic model composition and synthesis of C code from the composed model. We also describe a case study that applies Model-Driven Theme/UML to the design and implementation of a pacemaker.

The rest of this article is structured as follows. Section 2 describes relevant concepts in the Theme/UML aspect-oriented design language. Section 3 summarizes related work, while Section 4 illustrates by means of example how Theme/UML can aid reduction of complexity in embedded systems design. Section 5 describes the Model-Driven Theme/UML tool and associated development process, specifically focusing on its support for model-driven engineering of embedded systems. Section 6 presents an application development case study that illustrates the applicability of Model-Driven Theme/UML to the development of embedded systems. Section 7 concludes the article and outlines plans for future work.

2. THEME/UML

The Theme approach is an aspect-oriented methodology that encompasses the requirements analysis, design, and mapping to implementation phases of the development lifecycle [Clarke and Baniassad 2005a]. Theme/UML is an aspect-oriented modeling language that facilitates the graphical modeling of concerns in an extended version of UML. The accompanying methodology provides guidelines on the use of these constructs, which include a new type of classifier called a theme, and three integration strategies—merge, bind, and override.

A theme is a construct based on the standard UML package and encapsulates the design specification of a base or aspect concern. Figure 1 shows how base

[1]By advanced modularization, we mean separation of not only crosscutting concerns but also separation of functionally distinct base concerns.

themes and aspect themes are designed. Base themes are modeled using the standard UML process and any of the available diagram types. A base theme encapsulates a single piece of noncrosscutting functionality and can trigger the crosscutting behavior represented by an aspect theme. An aspect theme is one that encapsulates a crosscutting concern and is designed relative to abstract templates. In the current version of Theme/UML, sequence diagrams are used to specify when and how the templates interact with the base themes. However, composition relationships could equally be visualized using any of UML's standard behavioral diagrams including collaboration [Mehner et al. 2006], activity [Cui et al. 2009], and state [Cottenier et al. 2007a] diagrams. The choice of behavioral diagram to capture theme composition relationships depends largely on how behavior is modeled within individual themes. For example, if the base theme has modeled its behavior using state diagrams, then it is likely that it would be helpful to model corresponding compositions also using state diagrams [Cottenier et al. 2007b]. Theme/UML currently uses sequence diagrams as a source for synthesizing OO source code for languages such as Java. However, state diagrams are commonly used in event-driven, state-dependent embedded systems software design, and also lend themselves to transformation to source code. In future work, Theme/UML will be extended to support the use of a wider range of behavioral diagrams in the specification of compositions.

Themes in Theme/UML are declaratively complete, that is, each concern is self-contained and does not reference anything outside of its own specification. While this property facilitates a more rigorous separation of individual themes, it may result in the same concepts being simultaneously represented in multiple themes (though likely not specified in detail in more than one theme). Consequently, these overlapping concepts must be reconciled at composition time.

Theme/UML supports compositional constructs to cater for both overlapping and crosscutting relationships between themes [Clarke and Baniassad 2005b]. A merge integration strategy is used between two or more themes to produce a single theme containing the union of the merged themes. The merge integration strategy caters for overlapping concepts by allowing elements to be matched, for example, like-named elements, thereby resolving conflicts between themes. Figure 2 illustrates a merge between two themes, where the `match[name]` property indicates that elements are to be merged based on name. `ThemeName(''`*NewTheme*`'')` indicates that the merge will produce a new theme called *NewTheme*. To achieve resolution of conflicts, Theme/UML supports three reconciliation strategies. The first strategy, prec, indicates the precedence of each theme's design specification in the merge. Figure 2 illustrates that the second theme has a higher precedence than the first theme, and therefore, its design specification will get priority in the merge. The second reconciliation strategy is an explicit reconciliation that takes the form `resolve(Entity (property = value))`, and allows any property of any specific entity in a theme to be assigned a value. The third reconciliation strategy is a default reconciliation and has a similar form to the second strategy, with Construct replacing Entity (as illustrated by the possibility to include either Construct or Entity in

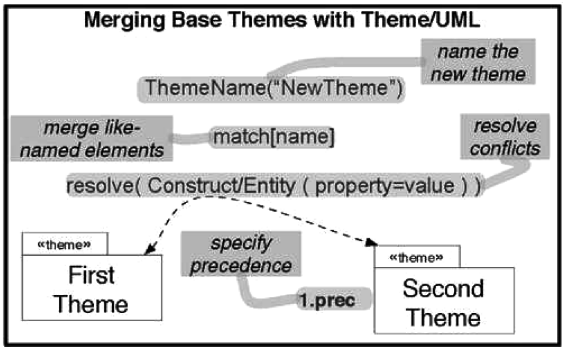

Fig. 2. Merge integration strategy.

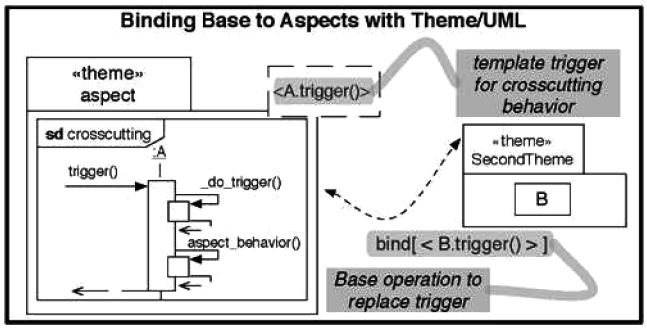

Fig. 3. Bind integration strategy.

the resolve statement shown in Figure 2). In this case, any property of a UML construct (e.g., operation visibility kind) can be given a value (e.g., private), with the reconciliation being executed during the merge.

A second integration strategy called a *bind* facilitates composition of aspect themes with base themes, for example, specification of how crosscutting embedded systems concerns are composed with core functional concerns. In Figure 3, an aspect theme is designed in relation to its abstract templates. The triggering template operation in this example is called `A.trigger()`. The sequence diagram illustrates the behavior of the aspect theme in relation to this triggering behavior. The operation `_do_trigger()` represents the behavior of any operation in the base theme that will be bound to the template method. The sequence diagram represents how and when the crosscutting behavior is executed with respect to the base themes that it affects. For example, in this case, the crosscutting behavior occurs after the execution of the bound operation. The bind specification represents instantiation of the aspect theme. The aspect theme is bound to the base operation `B.trigger()`; therefore, the triggering template operation is replaced with `B.trigger()` when the aspect is instantiated. This behavior allows crosscutting concerns to be modularized and subsequently composed with specific points in the base behavior of a system.[2]

[2]See Clarke [2001] for a more formal specification of the integration strategies.

Although not of particular relevance to this article, the third kind of integration strategy is called an *override* and deals specifically with overlapping specifications. An override is a relationship between two themes in which the design specification of one theme is denoted to override that of another.

3. RELATED WORK

In this section, we review a number of related approaches to model-driven engineering of embedded systems. We also briefly survey (i) alternative approaches to modularisation of concerns in design and (ii) the relationship of our work to existing approaches to formal specification, validation, and synthesis of embedded systems.

3.1 Model-Driven Engineering of Embedded Systems

Although model-driven engineering has been steadily gaining popularity since the introduction of CASE tools in the 1980s, the mismatch between UML (typically used as the modeling language in model-driven engineering environments) and embedded platforms (typically non–object-oriented) has hindered progress in applying MDE techniques to development of embedded systems. However, the concept of taking a more model-centric approach to developing embedded systems is rapidly gaining acceptance (as evidenced by Gérard et al. [2005]) and a number of related approaches have been proposed in recent years.

Telelogic Rhapsody Developer in C[3] is a model-driven engineering tool that generates C code based on UML 2.0 behavioral and structural diagrams. UML models in Rhapsody Developer conform to the FunctionalC UML Profile [Douglass 2008] discussed in Section 5.4 and code is produced based on the UML to C mapping defined by the profile. While Rhapsody Developer offers automatic transformation of FunctionalC models to 32-bit real-time operating system environments and 8/16-bit microcontrollers, it has a number of limitations. Unlike Model-Driven Theme/UML, Rhapsody Developer has no support for advanced modularization in design, and hence, it does not address the issue of increasing complexity in embedded systems.

Riccobene et al. [2006, 2009] have presented a development environment for the hardware/software co-design of embedded systems based on UML and SystemC. The approach supports graphical specification of SystemC designs in UML, generation of SystemC code from the UML models and the option to reverse engineer UML models from SystemC code. Like Rhapsody Developer, this tool does not provide any support for advanced separation of concerns in design.

Nascimento et al. [2006] have presented the implementation of a UML/MOF metadata repository that provides an API to manipulate stored metadata. This API facilitates implementation of tools for embedded systems design. The repository caters for three types of application model—applications modeled using UML, platforms modeled using the UML profile for scheduling,

[3]http://modeling.telelogic.com/products/rhapsody/software/developer/developer-c.cfm.

performance, and time (UML-SPT),[4] and mappings between applications and platforms. In order to illustrate the approach the authors describe an application development case study that involves specification of the system design with UML models annotated with UML-SPT and automatic generation of Java source code. Similar to the tools reviewed previously in this section, the work of Nascimento et al. does not consider support for advanced modularization in design.

Oliveira et al. [2007] describe a model-driven approach for MPSoC design space exploration that focuses on addressing the issue of reasoning about complex multiprocessor systems. The tool, which is partially based on the work of Nascimento et al. discussed previously, supports a UML-based approach to design space exploration and an automatic multiobjective design space estimation mechanism. Support for model transformation is provided following the exploration phase of the development cycle, although it is unclear whether automated support for code generation is provided. The tool is not explicitly concerned with separation of concerns in the manner that Model-Driven Theme/UML is.

The Distributed Embedded Real-time Aspects Framework (DERAF) provides an extensible set of aspects related to nonfunctional requirements in embedded systems that can be used as part of the modeling phase, where models are created using UML-SPT [Wehrmeister et al. 2007]. The primary goal of the framework is to provide aspects that can alter the behavior of modeled system elements by adding behavior and structure to handle nonfunctional requirements, without committing the model to a specific implementation. The aspects provided in the initial version of DERAF cater for nonfunctional requirements related to time, performance, distribution and embedded concerns. Within each category of concern the authors have defined a number of subconcerns, for example, response time and throughput are the nonfunctional requirements supported in the performance category. DERAF's primary aim is to manage the growing complexity of distributed, real-time embedded systems, an aim shared with the approach described in this article. However, unlike Model-Driven Theme/UML, DERAF does not propose model composition semantics, and consequently it does not support automated model composition and subsequent generation of source code.

3.2 Modularization of Concerns in Design

There are a number of existing approaches to advanced modularization in design. Composition directives [Reddy et al. 2006; Straw et al. 2004] is an approach implemented in Kermeta called Kompose [Fleurey et al. 2007] that supports composition of both aspect and base UML models. The Atlas Model Weaver (AMW)[5] is a tool based on the Eclipse modeling framework that facilitates the creation of links between models [Didonet Del Fabro et al. 2005]. The Motorola WEAVR [Cottenier et al. 2007a] is a commercial add-in to the

[4]http://www.omg.org/docs/formal/05-01-02.pdf.
[5]http://www.eclipse.org/gmt/amw.

Telelogic TAU tool, designed for use in telecoms systems engineering, that supports aspect-oriented extensions to state machines. XWeave is a model weaver that supports composition of different architectural viewpoints, that is, the weaver facilitates software product-line engineering by allowing for variable parts of architectural models to be woven according to a specific product configuration [Groher and Voelter 2007]. Modeling aspects using a transformation approach (MATA) [Jayaraman et al. 2007] is a UML aspect-oriented modeling tool that uses graph transformations to specify and compose aspects. Finally, Klein et al. [2007] suggest an approach for weaving multiple behavioral aspects using sequence diagrams.

The overriding difference between our Theme/UML-based approach and the approaches surveyed is our support for commonly used/standard tools and techniques and our focus on supporting embedded platforms. Model-Driven Theme/UML purposely supports standard tools and formats, for example, UML, Eclipse, XMI, in order to ease adoption of our systematic, integrated approach to model-driven engineering. Additionally, we explicitly target embedded platforms through the provision of automatic transformation of composed models to C code.

3.3 Formal Model Specification, Validation, and Synthesis

A significant amount of research has been published on model-based design of embedded systems. Edwards et al. [1997] surveyed numerous approaches to formal modeling, verification, and synthesis of reactive real-time embedded systems. More specific surveys on the array of methods and tools available for formal verification of embedded systems have also been published [Wang 2004; Loghi et al. 2005; Cheng 2006], while a recent survey by Gherbi and Khendek [2006] discusses various extensions to UML that support modeling of real-time systems. Developer adoption of these approaches is supported by a range of tools for embedded systems development. These tools, reviewed by Emerson [2004] and the Escher Research Institute [2008], address various aspects of model-based design, from heterogeneous modeling, simulation and design of concurrent, real-time, embedded systems with Ptolemy II,[6] to hardware/software co-design with POLIS.[7] We believe that Model-Driven Theme/UML complements many of these approaches and that there is scope for integrating Model-Driven Theme/UML with the appropriate existing approaches. For example, our toolset's formal modeling and verification capabilities can be enhanced by incorporating existing support for modeling real-time behavior with UML [OMG 2007] and verification of UML sequence diagrams [Aredo 2002] and class diagrams [Cabot et al. 2008], respectively. Additionally, there is scope to use Model-Driven Theme/UML to analyze and modularize crosscutting concerns in the output of tools (e.g., Rhapsody Developer, POLIS, and Ptolemy II), that support synthesis to C.

[6]http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm.
[7]http://embedded.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

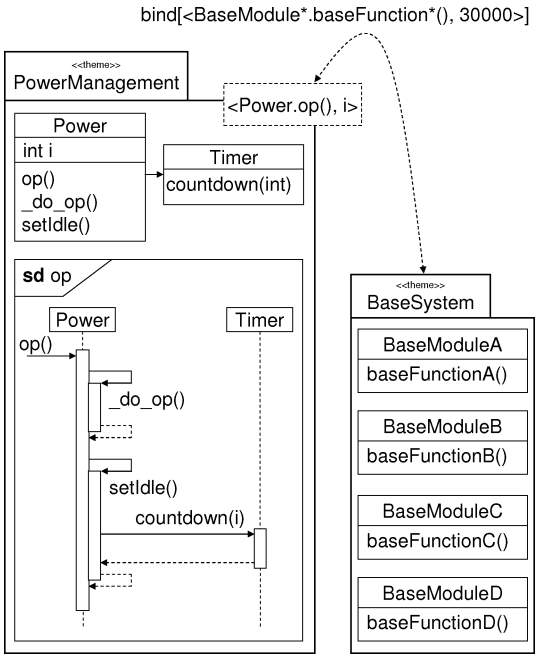

Fig. 4. The Theme/UML design of the power-saving concern applied to core system behavior.

4. SEPARATION OF EMBEDDED SYSTEMS CONCERNS WITH THEME/UML

Embedded systems have become increasing complex in recent years as improvements in hardware performance and reductions in cost have enabled construction of more technically advanced systems. One feature of this complexity is the repetition of nonfunctional behavior throughout a system (e.g., memory management, resource sharing, real-time and power management code). In this section, we illustrate by means of a power management example how Theme/UML can reduce the presence of repeated, crosscutting concerns in embedded systems design.

Figure 4 illustrates the Theme/UML design of a power management concern for a remote control (e.g., a wireless games console controller) that sleeps after a period of inactivity. Power management is orthogonal to the core behavior offered by the system to which it is being applied and is, therefore, modeled separately. The design contains two themes, an aspect theme on the left-hand side called `PowerManagement` and a generic base theme called `BaseSystem` that consists of a number of modules that contain functions with power management requirements.

The aspect theme, `PowerManagement`, contains both structural and behavioral information. The structure of the theme is represented by a class diagram. `Power` contains three functions: `op()`, `_do_op()`, and `setIdle(int)`. `op()` is a Theme/UML trigger template function that represents invocation of any function that is bound to the aspect theme. `_do_op()` is a Theme/UML template function that is replaced with the execution of the bound method. The

`setIdle(int)` function contains a power management code. This method takes a parameter specifying the amount of time after which the device should be put into idle mode, and it invokes the `countdown(int)` function in the `Timer` class.

The behavior of the aspect theme is represented by a sequence diagram. The diagram, named `op()`, specifies the sequence of actions that occur once the method bound to `op()` in the `Power` class is invoked. `_do_op()` is invoked following invocation of the bound method, meaning that the bound method executes as normal before the crosscutting behavior executes. When the bound method has finished executing, the `setIdle(int)` method that starts the timer is invoked. Once the timer has completed the device can be put into idle mode.

Once the design of the base theme and the aspect theme are complete they can be associated with each other. The top right-hand corner of the aspect theme contains template parameters in a dashed box. This box specifies the template methods and parameters that are replaced with actual constructs from whatever base themes are bound to the aspect theme. In this case, the aspect theme design specifies that `op()` in class `Power` is a template function, and also that the bound theme can supply a numerical argument representing the amount of time to wait before switching to idle mode. The bind tag, illustrated at the top of the figure, specifies the composition relationship that exists between the aspect theme and the base theme that is bound to it. In this example, the bind tag specifies that each function in each of the modules in the base theme should be augmented with power-saving behavior and that the system should switch to idle mode 30,000ms after the completion of each bound function.

The aspect-oriented design of the power management concern illustrates that, using Theme/UML, crosscutting nonfunctional requirements can be designed in isolation and subsequently associated with the base concerns they affect via a composition relationship. In the following section, we describe Model-Driven Theme/UML, a prototypical tool we have developed that supports specification of embedded systems designs with Theme/UML, automated model composition based on Theme/UML composition relationships and automated model-to-code transformation.

5. MODEL-DRIVEN THEME/UML

Model-Driven Theme/UML is a set of tools with a supporting development process that facilitates modularized design with Theme/UML and subsequent model composition and synthesis to source code. The tool suite and process was originally designed to support development of applications for deployment on mobile platforms [Carton et al. 2009]. However, we have extended the tool suite to support transformations to embedded platforms by offering synthesis to C code. Figure 5 illustrates the Model-Driven Theme/UML development process. The process contains three distinct phases that are titled based on the activities of the developer during each phase.

5.1 Modeling

The first phase involves system modeling via specification of base concerns, aspect concerns and composition relationships in Theme/UML. We

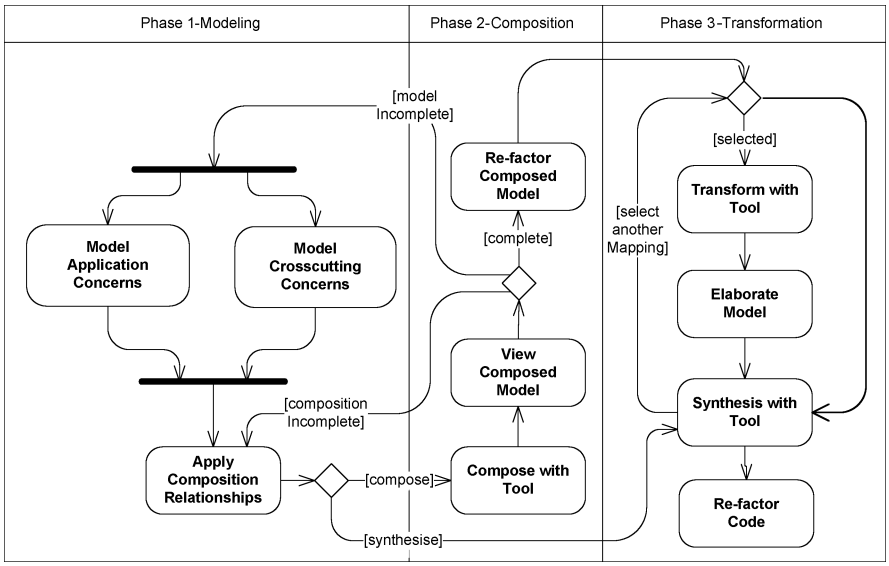

Fig. 5. Phases in the Model-Driven Theme/UML development process.

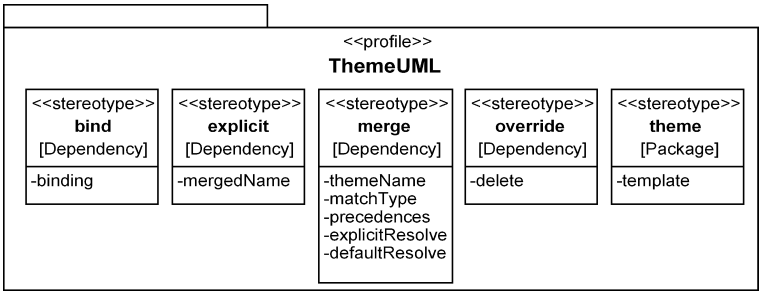

Fig. 6. The Theme/UML marking profile.

have developed a Theme/UML marking profile that facilitates development of Theme/UML models within standard UML editors. Marking allows for model elements to be tagged with information that serves as input to the model composition process (discussed later in this section). Figure 6 illustrates that the Theme/UML profile contains a theme stereotype extending the UML package construct and a series of relationships between themes based on UML dependency relationships. Developers couple this lightweight extension to UML with a graphical UML editor to create modularised Theme/UML system designs before proceeding to the model composition phase of the development process. The primary requirement for any UML editor used during the modeling phase is that it supports export to Eclipse modeling framework[8] (EMF) XML metadata interchange (XMI) format. Our current editor of choice is MagicDraw,[9] which is free under academic license. However, any UML editor that supports

[8]http://www.eclipse.org/modeling/emf/.
[9]http://www.magicdraw.com.

integration of profiles and export to XMI can be used during the modeling phase.

5.2 Composition

The second phase of the Model-Driven Theme/UML development process involves composition of the models specified during the modeling phase. By supporting model composition, we offer developers the benefits of using separation of concerns in design without restricting them to implementing their solutions in aspect-oriented languages. This is of particular relevance in the embedded systems domain where the use of aspect-oriented programming languages during implementation is uncommon. Implemented as an Eclipse plug-in, our composition tool takes the XMI produced during the modeling phase as input and processes it based on knowledge of the Theme/UML composition metamodel to produce a composed, platform-independent standard UML view of the system model.

At this point in the process, there is a decision point. Either the developer is satisfied with the results of the automatic composition or the composition is deemed incomplete. In the first case, the developer can elaborate the composed model and/or proceed to the next phase. In the case of an incomplete composition, the developer can revert to the modeling phase, reapply the Theme/UML composition relationships and compose the model again.

5.3 Transformation

The transformation phase begins with a decision regarding how the developer wishes to transform the composed model. Of primary relevance to this article is the arrow in Figure 5 from the decision point in the transformation phase to the state labeled "Synthesis with Tool," which illustrates that C code can be automatically generated from the composed model. We adopted two approaches for model-to-text transformation—a template-based approach for structural code and a visitor-based approach for behavioral code. The implementation of both these approaches is based on a third-party code generation technology called openArchitectureWare.[10] openArchitectureWare is a model-driven development generator framework that, via a template-based language called XPand, facilitates production of text from models. We developed a set of templates that encode a UML-to-C mapping, discussed in Section 5.4, and are processed by the openArchitectureWare workflow engine to produce C code from a UML model. We extended the templates with Java extensions that implement behavioral code generation from sequence diagrams and integrated them using openArchitectureWare's template extension mechanism. The result is an integrated code generation tool that produces compilable structural and behavioral code from class diagrams and sequence diagrams respectively.

[10]http://www.openarchitectureware.org/Now part of the Eclipse modeling, Model to Text Project http://www.eclipse.org/modeling/m2t/.

5.4 Mapping UML to C

UML is a general-purpose, programming language-independent modeling language that can be used to model a vast range of systems. However, due to its object-oriented nature, systems designed in UML map more naturally to implementation in object-based languages such as C++ and Java as opposed to procedural languages like C. Therefore, it was necessary to define a mapping from UML modeling constructs to C language constructs in order to facilitate code synthesis.

Previous work on mapping from UML to C suggests there are two primary options when defining a mapping. The first option is to define a new UML profile that effectively transforms UML from an object-based modeling approach to a functional modeling approach. Such an approach to using UML (and by implication, Theme/UML) is probably more natural to C programmers, in that it avoids unfamiliar object-oriented concepts such as inheritance. Modeling C applications in this way is consistent with the way many C designers design their applications today [Bakal and Cohen 2010]. For example, FunctionalC, developed by Telelogic, introduces new concepts to UML so that it can support developers in specifying functionally oriented, C-based systems [Douglass 2008]. In FunctionalC, a standard UML class diagram becomes a File diagram, where files are composed of variables and functions. Extending UML with a language-specific profile fosters use of UML by developers who are not familiar with UML design, allowing them to graphically describe their software without any prior knowledge of object-oriented modeling. However, many of the advantages of object-oriented modeling, such as the reduction in complexity due to the use of inheritance, are lost.

The second option when mapping UML to C is to forgo functionally oriented UML extensions, embrace the full power of object-oriented modeling and map object-oriented UML constructs to C language constructs. Khan et al. [2006] have proposed a mapping correspondence from UML 2.0 to C, and Pont [2002] indirectly proposes a mapping during a discussion on how to use an object-oriented style of programming with C. These mappings describe how UML/object-oriented constructs such as classes, their attributes and operations map to C language constructs.

Our current approach to mapping UML to C is a hybrid approach based on ideas introduced in the FunctionalC profile, where a File diagram replaces a Class diagram, and the mappings of Khan and Pont, where behavioral elements in standard UML classes are mapped to standard C language constructs. Table I describes the mapping we use to transform composed platform-independent models to C code. The mapping is realized as openArchitectureWare templates and extensions and is discussed in detail during the case study in Section 6. Our mapping from UML Class to C struct does not currently support the use of inheritance. To overcome this limitation, thereby more fully supporting object-orientation at the model level, our UML to C mapping needs to be extended. In this future work, we will use well-accepted mappings from generalization relationships in the model to standard C language constructs (See Section 6.1).

Table I. The UML to C Mapping Employed in Model-Driven Theme/UML

UML	C	Explanation
Class	File	Each class in a UML class diagram maps to a pair of C files in source code (a .C and a .H file). Constructor and destructor functions are included in the .C and .H files.
Class	Struct	In cases where multiple instances of a class are required, each class in a UML class diagram maps to a struct defined in a .H file.
Attribute	Variable	Each attribute of a class in a class diagram maps to a variable in a .C file. Private variables are defined as static. In the case of multiple instances, the attribute would map to a variable in a struct.
Operation	Function	Each public operation of a class in a class diagram maps to a function declared in the .H and .C files. Private functions are defined as static in the .C file only.
Association	Header Include	Association relationships between classes in a class diagram map to the inclusion of external header files within a file.
Sequence Message	Function call	Messages in UML sequence diagrams map to sequences of C function calls.

6. CASE STUDY

In order to assess the applicability of Model-Driven Theme/UML to embedded application development, we conducted a case study involving the design and implementation of an artificial pacemaker. The case study illustrates how Model-Driven Theme/UML facilitates separation of concerns in design and both automated model composition and source code generation.

A pacemaker is a medical device that uses electrical impulses to regulate the beating of the heart. The pacemaker under consideration has typical requirements such as sensing the heart rate and generating a pulse. It is also required to wirelessly communicate diagnostic information and support updates.

Traditional approaches to developing such a system begin with a requirements specification that is directly implemented as code by hand. Rapid prototypes may be produced to learn about requirements or an intermediate modeling step may be taken using tools and environments such as MATHLAB/Simulink, Labview, or Ptolemy [Kopetz 1997; Lee 2003; Shukla 2009]. Other model-driven approaches include Telelogic Rhapsody Developer and the SystemC development environment by Riccobene et al. [2006, 2009] (see Section 3). These tools can be used to produce physical or electrical models and can output production C code or generate hardware description language code for FPGA devices [Krasner 2004]. The design then goes through a verification stage, which may be formal or simulation based (possibly with hardware in the loop) [Murphy et al. 2008].

In contrast, the Theme/UML approach illustrated by this case study, supports correct-by-construction software synthesis through an aspect-oriented system decomposition and model-driven development process [Shukla 2009]. This aspect-oriented approach provides a stronger separation of concerns capability, allowing the designer reason about different functionalities, including those that scatter across a system, separately. This approach complements existing approaches to embedded system development by facilitating

Table II. Themes in the Pacemaker Design

Base Themes	Functional Requirement
Monitor Heart	Provide access to heartbeat and heart temperature sensor data.
Analyze Pulse	Analyze the frequency of the heartbeat to determine the current heart rate.
Generate Pulse	Adjust the heart rate based on knowledge of the ideal rate and the current rate.
Acquire Data	Provide remote, wireless access to the pacemaker and access to diagnostic information.
Update Pacemaker	Remotely update the pacemaker firmware.
Aspect Themes	Nonfunctional Requirement
Security	Provide data security on outgoing communications via data encryption. The security aspect theme has a trigger method called `transmit()`. Any function bound to this trigger template method has encryption applied to the messages it sends before they are transmitted.
Memory Management	Provides functions to reduce memory allocation problems. The memory management aspect has two trigger template methods called `create()` and `destroy()`. The aspect's behavior checks pointers, prevents memory leaks, stops buffer under- / overruns and ensures newly allocated buffers contain a well-known value.
Fault Tolerance	Cater for sensor errors by coordinating between multiple sensor readings. The fault-tolerance aspect theme has a trigger template method `sense()`. Comparison analysis and coordination behavior is applied following execution of functions bound to the trigger method.

separation of concerns in design, automated model composition, and source code generation. The use of models has the potential to facilitate verification at the model level, thus overcoming the state–space explosion problem related to formally verifying code [Pelánek 2009; Shukla 2009].

The Model-Driven Theme/UML case study began with the identification of five base themes and three aspect themes. The base themes encapsulate the core system behavior, while the aspect themes represent features that crosscut the base themes. Table II lists the themes identified and outlines their responsibilities.

In accordance with the Model-Driven Theme/UML process illustrated in Figure 5, application development began with modeling of base and crosscutting concerns in the UML editor. In the interest of brevity, we do not show and discuss the design of all themes, but rather we focus on a number of representative examples.

Figure 7 shows the structural design of `GeneratePulse`, an example of a base theme created during the design of the pacemaker.[11] The theme contains only the classes and relationships necessary to represent the structure of the

[11]Illustration of the detailed design is outside the scope of this section. However, base themes can be specified in greater detail by including one or more behavioral diagrams. Additionally, there is scope to integrate complementary UML profiles for modeling embedded systems behavior (e.g., MARTE [OMG 2007].

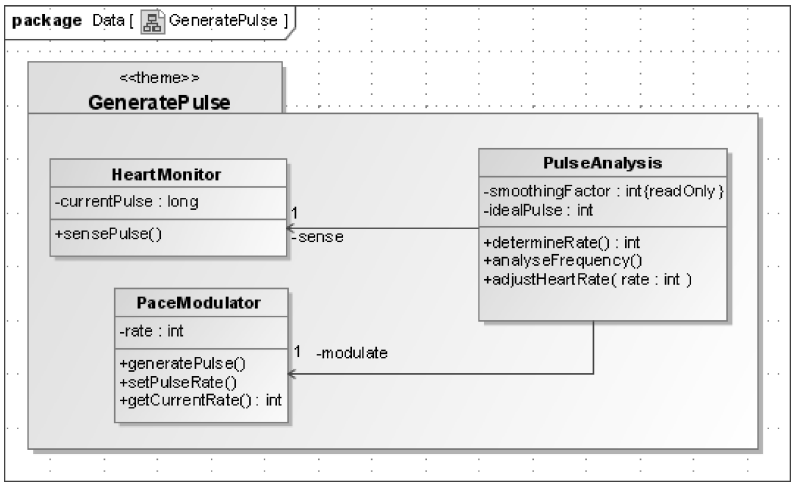

Fig. 7. Screenshot of the Magicdraw design of the `GeneratePulse` base theme.

pulse generation behavior (i.e., generation of a heart pulse based on analysis of the current heart rate. Figure 8 illustrates the design of both the structural and the behavioral constituents of the `MemoryManagement` aspect theme. `MemoryManagement` supports dynamic memory allocation by keeping track of all allocation and frees.[12] Functions that allocate or free memory are bound to one of the triggering template operations: `create()` or `destroy()`. These functions wrap standard memory allocations with checks designed to prevent problems such as memory leaks, dependence on values in newly allocated buffers, pointer over- and underruns, and invalid pointers. The design illustrates that the crosscutting memory management concern is cleanly modularized within the aspect theme as opposed to occurring repeatedly at relevant points throughout the code base.

When all of the base and aspect themes had been designed, we were in a position to specify composition relationships between them. The relationships dictate how the base themes are composed to form a complete system and how the base themes are bound to the aspect themes that crosscut them. Figure 9 shows a screenshot of the theme composition diagram as designed in Magicdraw. The composition specification illustrates that all base themes should be merged together to form a system called "Pacemaker," as denoted by the `ThemeName` tag on the merge relationship beneath the `AcquireDataExternally` theme. This merge relationship also contains a tag that specifies that identically named concepts in different themes should be considered to represent the same concept and should be merged. The theme composition diagram also shows how base themes are bound to aspect themes. `sendMessage()` in the `Comms` class of the `AcquireDataExternally` theme is bound to the `Security` aspect theme so that data sent from the pacemaker to the outside world is encrypted. All sensing methods in the `MonitorHeart` theme are bound to the `DiversityFaultTolerance`

[12]The behavior of the memory management aspect is based on the features provided by Walter Bright's MEM Package [Bright 1990].

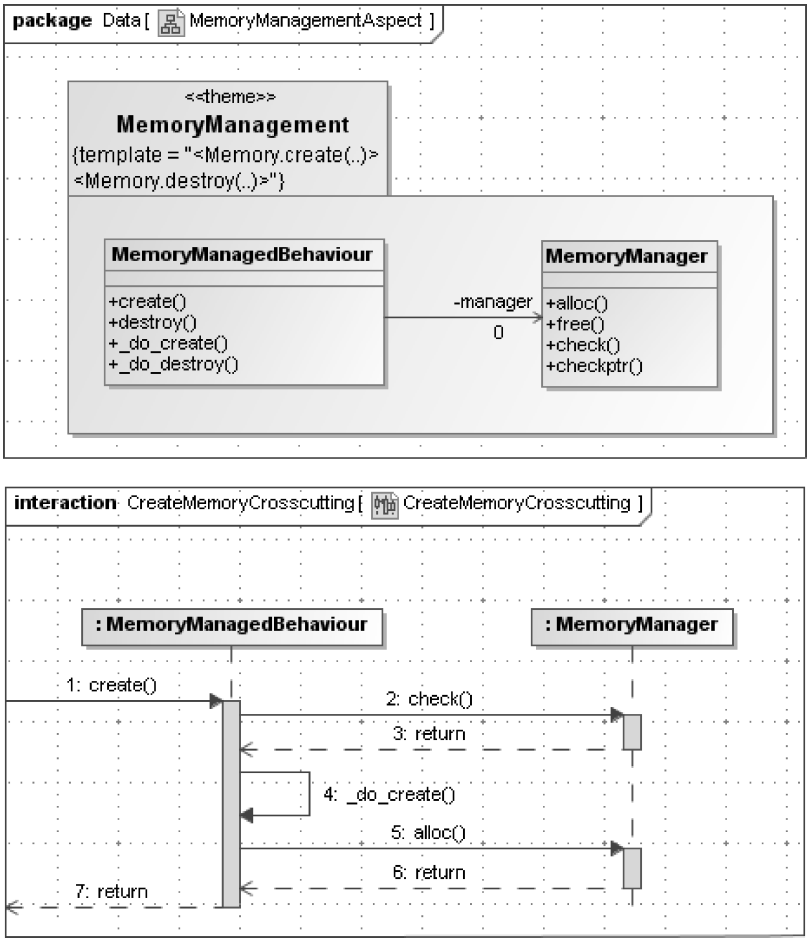

Fig. 8. Screenshots of the Magicdraw design of the `MemoryManagement` aspect theme.

aspect so that diversity in sensor readings can be detected and inaccurate readings discarded. Methods in `MonitorHeart` and `AcquireDataExternally` that allocate or free memory, for example, `receiveMessage()` and `sendMessage()` in the `Comms` class and `init()` in all the `HeartMonitor` classes, are bound to the `MemoryManagement` theme.

When the composition specification was completed, the full pacemaker design (base themes, aspect themes, and theme composition diagram) was exported from Magicdraw to Eclipse in XMI format (EMF UML2 v2.x XMI File). The XMI version of the pacemaker design, shown in Figure 10, is used as input to the Theme/UML model composition plug-in. The plug-in uses the composition relationships specified during the design process to output a composed version of the pacemaker system in XMI format. The composed model can be viewed graphically as a class diagram to aid the process of examining the result of the composition. Figure 11 shows a screenshot of the composed version of the pacemaker design.

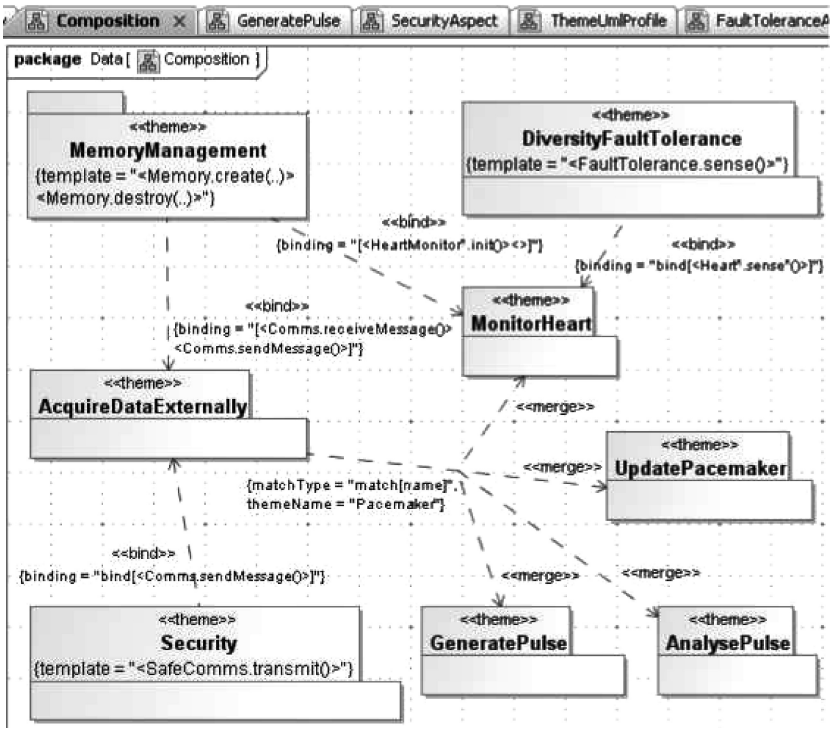

Fig. 9. Screenshot of the theme composition diagram as designed in Magicdraw.

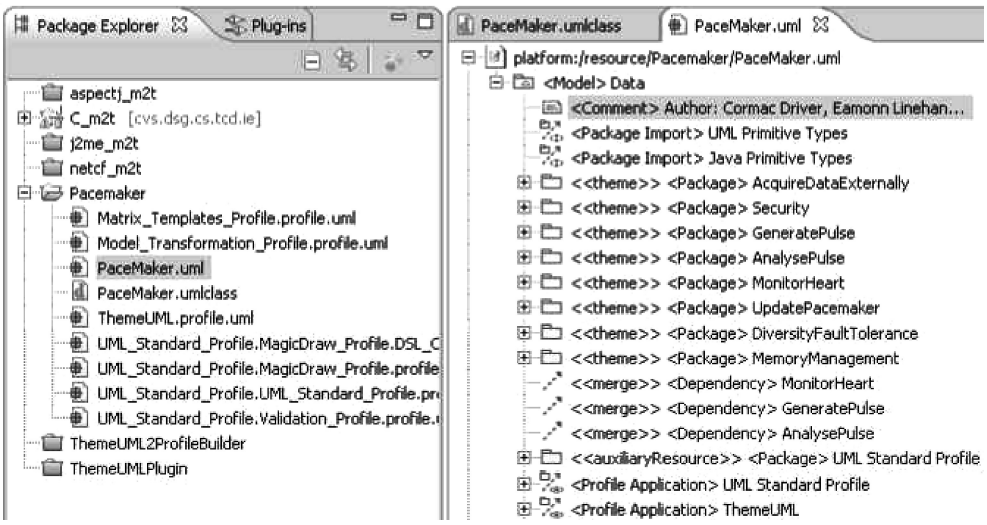

Fig. 10. Eclipse screenshot of the pacemaker design in XMI format.

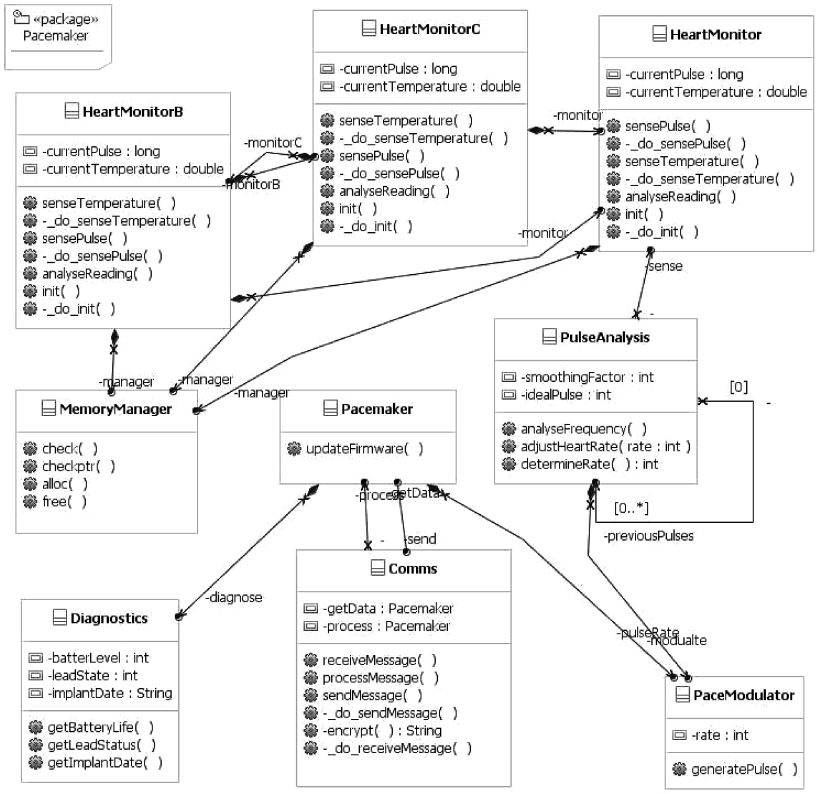

Fig. 11. Screenshot of the composed pacemaker design as viewed using UML2 Tools in Eclipse.

The figure shows that the composition relationships have been applied to produce a composed UML version of the pacemaker design. The crosscutting behaviors specified in the aspect themes and bound via bind statements in the composition diagram have been woven and are represented by the presence of pairs of similarly named methods in the affected classes, for example, the `Comms` class has methods `sendMessage`, and `_do_sendMessage`, as it is affected by the memory management concern.

With the composition process complete it was possible to automatically generate source code from the UML model. This was achieved using templates and extensions built using the openArchitectureWare framework. The templates encode a mapping from UML to C based on the mapping discussed in Section 5.4. Figure 12 contains a screenshot of `Class.xpt`, a template written in XPand that specifies how each class in the composed model should be treated by the openArchitectureWare workflow engine. The screenshot illustrates that for each UML class encountered a .c and a .h file are created, both named after the name of the class. In the case of the .c file, the attribute, static attribute and operation templates are then applied (lines 12 through 14 in Figure 12). For .h files, the header attribute and header operation templates are applied (lines 25 and 26, respectively). The templates generate the

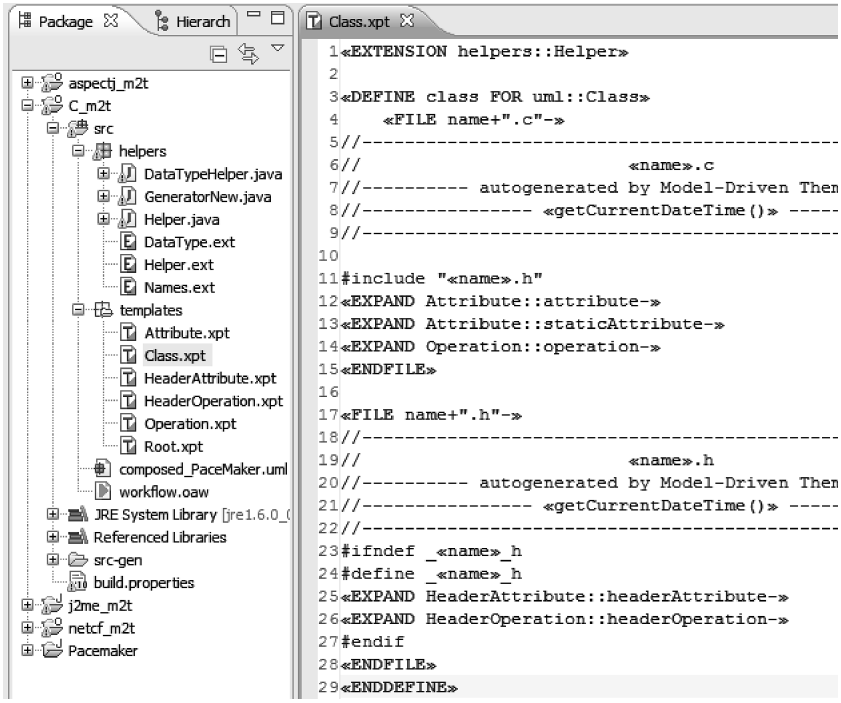

Fig. 12. Screenshot of the `Class.xpt` XPand template.

structural code, while the Java class `GeneratorNew.java` (shown to the left of lines 7 and 8) contains model parsing code to generate behavioral code from sequence diagrams.

The code generation templates and extensions are processed by the openArchitectureWare workflow engine, which is invoked by running a workflow file. Figure 13 shows the workflow file that specifies the files that are involved in code generation for the pacemaker application. Line 5 specifies that the target of the workflow engine is the composed model produced by the Theme/UML Eclipse plug-in. The output produced is placed within a folder called src-gen (line 16) and the first code generation template applied to the model is `Root.xpt`, which identifies classes within packages and applies the `Class.xpt` template to each (as discussed in relation to Figure 12). The result of executing the workflow file is a .c and .h file for each class in the composed pacemaker model. Figure 14 shows code generated for the `PaceModulator.c` file. The code shown is predominantly structural due to the nature of the pacemaker design, which consisted mainly of class diagrams. As mentioned previously, specification of behavior in base themes is possible through the inclusions of behavioral diagrams (e.g., sequence diagrams) and results in generation of sequences of functions calls within method bodies. The `Comms_init()` function (lines 133 through 140) is affected by the memory management concern and consequently has crosscutting code woven into the function body.

workflow.oaw

```
<workflow>
    <bean class="oaw.uml2.Setup" standardUML2Setup="true" />

    <component class="oaw.emf.XmiReader">
        <modelFile value="ANSI-C_composed_PaceMaker.uml" />
        <outputSlot value="model" />
    </component>

    <component id="generator" class="oaw.xpand2.Generator"
        skipOnErrors="false">
        <metaModel class="oaw.type.emf.EmfMetaModel">
            <metaModelPackage value="org.eclipse.emf.ecore.EcorePackage" />
        </metaModel>
        <metaModel class="oaw.uml2.UML2MetaModel" />
        <expand value="templates::Root::Root FOR model" />
        <outlet path="src-gen/" />
        <prSrcPaths value="src-gen" />
        <prDefaultExcludes value="true" />
    </component>
</workflow>
```

Fig. 13. Screenshot of the openArchitectureWare workflow file for the pacemaker application.

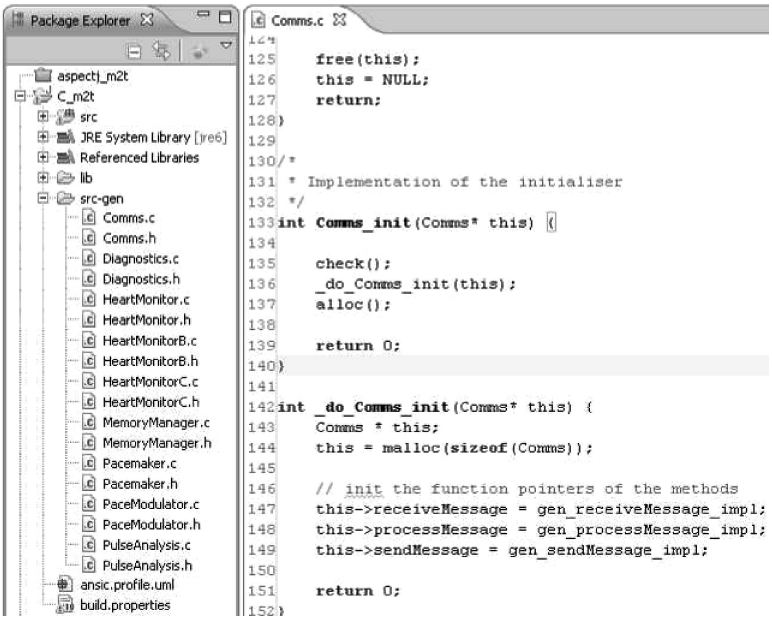

Fig. 14. Screenshot of the generated code for the `Comms.c` file.

6.1 Applicability to Embedded Systems

This case study illustrates how Model-Driven Theme/UML supports aspect-oriented model-driven engineering of embedded systems. Theme/UML facilitated separation of concerns at design time, allowing specific requirements to be reasoned about in isolation. This separation of concerns avoided the presence

of scattered crosscutting concerns throughout the design, helping to preserve key system properties such as comprehensibility, extensibility, and reliability. Given a set of composition relationships the Model-Driven Theme/UML tool automatically generated a composed view of the application, allowing progress from an aspect-oriented modeling environment to an object-oriented modeling environment. Automated source code generation facilitated a final transformation to a procedural programming environment. The tool supports a solution-focused development approach that allows developers to concentrate on modularising system design and to avail of the benefits of automatic model composition and source code generation.

However, the current version of Theme/UML has some limitations, some of which have already been identified throughout this article. First, the Theme/UML profile does not capture all embedded systems concerns. For example, Theme/UML does not support the specification of temporal behavior [Galpin et al. 2009]. We are currently investigating overcoming such limitations by specifying a model that inherits from more than one UML profile, for example, MARTE for temporal behavior [OMG 2007]. This will require extensions to the code generation aspects of the Theme/UML toolset. Second, the toolset does not support inheritance in the UML to C transformations, which precludes the use of inheritance at the modeling level. We are currently investigating the implementation of inheritance for C using an approach such as namespace management [Samek 1997] or message maps and aggregation [Sadler 1998]. Third, the current Theme/UML toolset implementation requires that composition behavior is specified in sequence diagrams. However, particularly in embedded systems, state diagrams are likely to be used in the base themes. Where the base theme's behavior is specified in state diagrams, it is probable that it would be useful to specify (at least some of) the compositional behavior also using state diagrams. This is also true for activity diagrams and collaboration diagrams. The inclusion of support for more than one behavioral diagram is planned as part of future development of the Model-Driven Theme/UML toolset.

Aside from these limitations, we envisage that Theme/UML will be used as a complementary approach with other modeling techniques. Theme/UML's strength is in managing complexity through the modularization of functional and nonfunctional concerns that, in particular, manifest themselves as code in the system (including crosscutting behavior). For nonfunctional concerns that do not give rise to explicit executable behavior, other approaches should be used. For example, standard OCL can be used for specifying constraints.

7. SUMMARY AND FUTURE WORK

In this article, we have presented our approach to aspect-oriented model-driven engineering for embedded systems. We illustrated how Theme/UML can be used to modularize nonfunctional embedded systems requirements. We followed that by describing Model-Driven Theme/UML, a toolset and supporting development process for separation of embedded systems concerns in design and subsequent automated model composition and C source code generation.

We also illustrated the capabilities of the tool by means of a case study and discussed related work.

We are currently investigating two main extensions to Model-Driven Theme/UML.[13] First, we are investigating embedded systems-specific extensions to the Theme/UML specification so that it will provide better support for specification of formal system models, which will in turn lend themselves to verification. Second, we are exploring how the output of existing model-based tools for embedded systems design can be reverse engineered to XMI and analysed from a software modularization perspective using Model-Driven Theme/UML.

REFERENCES

AFONSO, F., SILVA, C., MONTENEGRO, S., AND TAVARES, A. 2007. Applying aspects to a real-time embedded operating system. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, New York.

AREDO, D. B 2002. A framework for semantics of UML sequence diagrams in PVS. *J. Universal Comput. Sci. 8*, 7, 674–697.

BAKAL, M. AND COHEN, J. 2010. Modeling c applications in uml with files and structures. http://www.dsp-fpga.com/articles/id/?3637.

BRIGHT, W. 1990. Debugging c pointers using mem. *Micro Cornucopia*, 21.

CABOT, J., Clarisó, R., AND RIERA, D. 2008. Verification of UML/OCL class diagrams using constraint programming. In *Proceedings of the International Conference on Software Testing Verification and Validation Workshop.* IEEE, Los Alamitos, CA, 73–80.

CARTON, A., DRIVER, C., JACKSON, A., AND CLARKE, S. 2009. Model-driven theme/UML. *Trans. Aspect-Oriented Software Dev. VI 5560*, 238–266.

CHENG, A. M. K. 2006. A survey of formal verification methods and tools for embedded and real-time systems. *Int. J. Embedded Syst. 2*, 3/4, 184–195.

CLARKE, S. 2001. Composition of object-oriented software design models. Ph.D. thesis, DCU.

CLARKE, S. AND BANIASSAD, E. 2005a. *Aspect-Oriented Analysis and Design: The Theme Approach*, 1st ed. Addison-Wesley, Upper Saddle River, NJ.

CLARKE, S. AND BANIASSAD, E. 2005b. *Theme Composition*. Object Technology Series. Addison-Wesley, Boston, MA, 199–244.

COTTENIER, T., VAN DEN BERG, A., AND ELRAD, T. 2007a. The Motorola WEAVR: Model weaving in a large industrial context. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, Industry Track*. ACM, New York.

COTTENIER, T., VAN DEN BERG, A., AND ELRAD, T. 2007b. Stateful aspects: The case for aspect-oriented modeling. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*. ACM, New York, 7–14.

CUI, Z.,WANG, L., AND LI, X. 2009. Modeling and integrating aspects with uml activity diagrams. In *Proceedings of the 2009 Symposium on Applied Computing.* ACM, New York, 430–437.

DENG, G., SCHMIDT, D. C., AND GOKHALE, A. 2006. Addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems via aspect-oriented and model-driven software development. In *Proceeding of the 28th International Conference on Software Engineering*. ACM, New York, 811–814.

DIDONET DEL FABRO, M., BÉZIVIN, J., JOUAULT, F., BRETON, E., AND GUELTAS, G. 2005. AMW: A generic model weaver. In *Journées sur l'Ingénierie Dirigée par les Modμeles (IDM05)*. 105–114.

DIJKSTRA, E. W. 1976. *A Discipline of Programming*. Prentice Hall, Upper Saddle River, NJ.

DOUGLASS, B. P. 2008. UML for C. White paper, Telelogic.

EDWARDS, S., LAVAGNO, L., LEE, E., AND SANGIOVANNI-VINCENTELLI, A. 1997. Design of embedded systems: Formal models, validation, and synthesis. *Proc. IEEE 85*, 3, 366–390.

EMERSON, M. 2004. A survey of embedded systems tools. *SIGBED Rev. 1*, 3, 3–5.

[13]The tools and examples presented in this article along with information on on-going work are available on our Web site http://www.dsg.cs.tcd.ie/aspects/themeUML.

ESCHER. 2008. Tools - Escher Research Institute. http://www.escherinstitute.org/Plone/tools/.

FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M. 2005. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, MA.

FLEUREY, F., BAUDRY, B., FRANCE, R., AND GHOSH, S. 2007. A generic approach for automatic model composition. In *Proceedings of the Aspect Oriented Modeling Workshop*. ACM, New York.

GALPIN, D., DRIVER, C., AND CLARKE, S. 2009. Modelling hardware verification concerns specified in the e language: an experience report. In *Proceedings of the International Conference on Aspect-Oriented Software Development, Industry Track*. ACM, New York, 207–212.

GÉRARD, S., BABAU, J.-P., AND CHAMPEAU, J., EDS. 2005. *Model Driven Engineering for Distributed Real-time Embedded Systems*. Hermes Science and Lavoisier, Paris.

GHERBI, A. AND KHENDEK, F. 2006. UML Profiles for real-time systems and their applications. *J. Object Tech. 5*, 4, 149–169.

GLINZ, M. 2007. On non-functional requirements. In *Proceedings of the 15th International Requirements Engineering Conference*. IEEE, Los Alamitos, CA.

GREENWOOD, P., BARTOLOMEI, T., FIGUEIREDO, E., DOSEA, M., GARCIA, A., CACHO, N., SANTANNA, C., SOARES, S., BORBA, P., KULESZA, U., AND RASHID, A. 2007. On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of the Object-Oriented Programming Conference*. Springer-Verlag, Berlin.

GROHER, I. AND VOELTER, M. 2007. XWeave: models and aspects in concert. In *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*. ACM, New York, 35–40.

JAYARAMAN, P. K., WHITTLE, J., ELKHODARY, A. M., AND GOMAA, H. 2007. Model composition in product lines and feature interaction detection using critical pair analysis. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*. Springer-Verlag, Berlin, 151–165.

KHAN, M. U., GEIHS, K., GUTBRODT, F., GÄOHNER, P., AND TRAUTER, R. 2006. Model-driven development of real-time systems with UML 2.0 and C. In *Proceedings of the 3rd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE, Los Alamitos, CA.

KICZALES, G., Lamping, J., Menhdhekar, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, Berlin, 220–242.

KLEIN, J., FLEUREY, F., AND JZQUEL, J.-M. 2007. Weaving multiple aspects in sequence diagrams. *Trans. Aspect Oriented Software Dev. 3*, 167–199.

KOPETZ, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Norwell, MA.

KRASNER, J. 2004. Model-based design and beyond: Solutions for today's embedded systems requirements. Analyst report, American Technology International.

LADDAD, R. AND ALEXANDER, R. 2003. Quality time. *IEEE Software 20*, 6, 90–93.

LEE, E. A. 2003. Overview of the Ptolemy Project. Tech. rep. UCB/ERL M03/25, University of California, Berkeley, California.

LOGHI, M., MARGARIA, T., PRAVADELLI, G., AND STEFFEN, B. 2005. Dynamic and formal verification of embedded systems. *Int. J. Parallel Program. 33*, 6, 585–611.

MEHNER, K., MONGA, M., AND TÄANTZER, G. 2006. Interaction analysis in aspect-oriented models. In *Proceedings of 14th International Requirements Engineering Conference*. IEEE, Los Alamitos, CA, 66–75.

MILLER, J. AND MUKERJI, J. 2003. MDA Guide. Tech. rep., Object Management Group.

MUNNELLY, J., FRITSCH, S., AND CLARKE, S. 2007. An aspect-oriented approach to the modularisation of context. In *Proceedings of the 5th International Conference on Pervasive Computing and Communications*. IEEE, Los Alamitos, CA, 114–124.

MURPHY, B., WAKEFIELD, A., AND FRIEDMAN, J. 2008. Best practices for verification, validation, and test in model-based design. Tech. rep. 2008-01-1469, The MathWorks, Inc.

NASCIMENTO, F. A. M., DA S. OLIVEIRA, M. F., WEHRMEISTER, M. A., PEREIRA, C. E., AND WAGNER, F. R. 2006. MDA-based approach for embedded software generation from a UML/MOF repository. In *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*. ACM, New York, 143–148.

OLIVEIRA, M. F. S., EDUARDO W. B., NASCIMENTO, F. A., AND WAGNER, F. R. 2007. Model driven engineering for MPSOC design space exploration. In *Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design.* ACM, New York, 81–86.

OLIVER, I. 2003. Model driven embedded systems. In *Proceedings of the 3rd International Conference on Application of Concurrency to System Design.* IEEE, Los Alamitos, CA.

OMG. 2007. A UML profile for MARTE. Tech. rep. ptc/07-08-04, Object Management Group.

PELÁNEK, R. 2009. Fighting state space explosion: Review and evaluation. In *Proceedings of the Formal Methods for Industrial Critical Systems Workshop.* Springer-Verlag, Berlin, 37–52.

PONT, M. J. 2002. *Embedded C.* Addison Wesley, Upper Saddle River, NJ.

REDDY, Y. R., GHOSH, S., FRANCE, R. B., STRAW, G., BIEMAN, J. M., MCEACHEN, N., SONG, E., AND GEORG, G. 2006. *Directives for Composing Aspect-Oriented Design Class Models*. Springer-Verlag, Berlin.

RICCOBENE, E., SCANDURRA, P., BOCCHIO, S., ROSTI, A., LAVAZZA, L., AND MANTELLINI, L. 2009. Systemc/c-based model-driven design for embedded systems. *ACM Trans. Embedded Comput. Syst. 8*, 4, 1–37.

RICCOBENE, E., SCANDURRA, P., ROSTI, A., AND BOCCHIO, S. 2006. A model-driven design environment for embedded systems. In *Proceedings of the 43rd Annual Conference on Design Automation.* ACM, New York, 915–918.

SADLER, J. 1998. Object oriented idioms in c. http://ficl.sourceforge.net/oo_in_c.html.

SAMEK, M. 1997. Portable inheritance and polymorphism in c. *Embedded Syst. Program.*

SCHMIDT, D. C. 2006. Model-driven engineering. *IEEE Comput. 39*, 2, 25–31.

SELVARAJ, H. AND JOZWIAK, L. 2005. Reconfigurable embedded systems: Synthesis, design and application. *J. Syst. Archit. 51*, 347–349.

SHUKLA, S. K. 2009. Model-driven engineering and safety-critical embedded software. *Computer 42*, 9, 93–95.

SOUSAN, W., WINTER, V., ZAND, M., AND SIY, H. 2007. ERTSAL: A prototype of a domain-specific aspect language for analysis of embedded real-time systems. In *Proceedings of the 2nd Workshop on Domain-specific Aspect Languages.* ACM, New York, 1.

STRAW, G., GEORG, G., SONG, E., GHOSH, S., FRANCE, R. B., AND BIEMAN, J. M. 2004. *Model Composition Directives*. Springer, Berlin, 84–97.

WANG, F. 2004. Formal verification of timed systems: A survey and perspective. *Proc. IEEE 92*, 8, 1283–1305.

WEHRMEISTER, M. A., FREITAS, E. P., PEREIRA, C. E., AND WAGNER, F. R. 2007. An aspect-oriented approach for dealing with non-functional requirements in a model-driven development of distributed embedded real-time systems. In *Proceedings of the 10th International Symposium on Object and Component-Oriented Real-Time Distributed Computing.* IEEE, Los Alamitos, CA, 428–432.

ZHANG, C. AND JACOBSEN, H.-A. 2003. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development.* ACM, New York, 130–139.

Received September 2008; revised October 2009; accepted May 2010