A Vision for Integration of Embedded System Properties Via a Model-Component-Aspect System Architecture[1]

Christopher D. Gill

Department of Computer Science and Engineering
Washington University, St.Louis
cdgill@cse.wustl.edu

Abstract *– Current approaches to developing complex embedded systems, particularly those with constraints in addition to their functional requirements, suffer significant limitations when moving from system requirements to implementation. Chief among these limitations is the inability of current development approaches to achieve an appropriate match between the sets of abstractions in their programming models and the inherent structure of concerns that appears in modern embedded systems.*

While emerging techniques such as model-integrated computing, component middleware and aspect-oriented programming address parts of this problem, how they are to be combined for greatest effect is still an open research problem. This paper outlines the problem of requirements-driven integration of multiple embedded system properties within implementation software, and offers an architectural vision for system software in support of requirements-driven development of embedded systems.

Keywords *– distributed real-time and embedded systems, generative programming*

A. INTRODUCTION

Complex large-scale systems, especially those with constraints on timeliness, footprint, or other properties outside the *functional* semantics of the application, are posing an increasing challenge to current approaches to software and system engineering. Specifically, developing *correct* distributed real-time and embedded (DRE) systems requires programmers to address constraints in two distinct semantic dimensions:

- *Functional* constraints – algorithmic correctness, type safety, computability and similar concerns related to computations and their results.
- *Para-functional* [1] constraints – end-to-end timeliness, recovery from faults, memory footprint, security, concurrency, and similar concerns that fall outside the functional semantics of the system, but are nonetheless essential to its correct operation.

Not only do these dimensions invite different styles of programming for configuring their semantics correctly, but they often interact in ways that can cause them *interfere* with each other. This paper considers the fundamental problem of interference in systems programming, examines several existing approaches to address that problem, and proposes a solution architecture that both synthesizes and extends current approaches.

This paper is structured as follows. Section B presents the problem of interference, which is the motivation for this work. Section C examines the challenge of supporting appropriate abstractions in the face of an overall constraint structure that resists decomposition along a single dimension of abstraction. Section D examines current approaches to addressing interference between system aspects, and notes key related research problems that remain open today. To address the open problems described in Section D, Section E presents an integrated architectural vision that combines and augments the existing approaches of model-integrated computing, component middleware, and middleware aspect frameworks. Finally, Section F presents conclusions and describes future work.

B. MOTIVATION: INTERFERENCE

This section describes the problem of interference between system aspects, and gives examples of interference within the context of concurrent distributed component middleware. Section B.a first describes a simple motivating example in which such interference can occur. Section B.b explains in detail how functional and para-functional system aspects can interfere in the example given in Section B.a. Section B.c then discusses the more general subject of interference, and the need for further research in that area.

[1] This work was supported in part by the DARPA PCES program, under contract F33615-00-C-3048 to Boeing and contract F33615-03-C-4110 to Washington University.

a. A Motivating Example

Consider the class of systems consisting of interacting application *components* distributed across multiple embedded endsystems. In this paper the term "component" is used in its formal technical sense, *i.e.*, consisting of objects implementing specified component interfaces, *e.g.*, as defined by the CORBA Component Model (CCM) [2] or J2EE [3] standards.

For these kinds of embedded applications, standards-based middleware such as CCM or J2EE allows application-specific components to be plugged into more general middleware frameworks, allowing application-specific configuration and re-use of independently developed software components. Examples of applications that can benefit from this approach include industrial control, avionics mission computing, and automotive information systems.

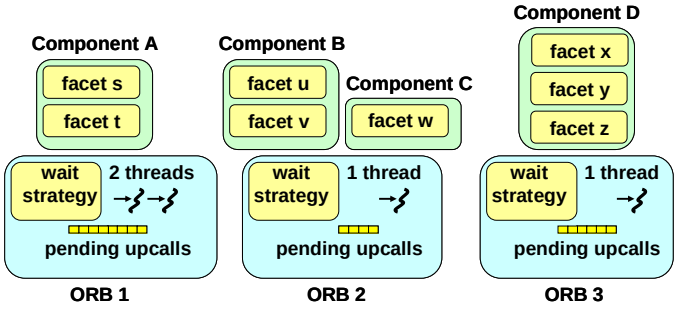

Figure 1: Deployment of Application Components

Figure 1 illustrates many key features of component-oriented embedded systems, including:

- *Components* – compose interfaces with objects that implement them;
- *Component Interfaces* – formally specify interactions between components:
  - *Facets* – advertise method invocation entry points for a component,
  - *Receptacles* – advertise points where invocations are made into other components,
  - *Event sinks* – advertise event push entry points for a component, and
  - *Event sources* – advertise event pushes into other components;
- *Supporting Infrastructure* – enables interactions between components, including:
  - Distributed communication – *e.g.*, between components on different object request brokers (ORBs), and
  - Local concurrency – *e.g.*, the number of threads available and the upcall concurrency policies in an ORB.

Use of component middleware technologies can significantly reduce the complexity of packaging, assembling, and deploying application components, but unfortunately are mainly focused on the *functional* properties of the components, *i.e.*, their implementation and interfaces but not how quality of service (QoS) requirements are communicated to or enforced in the underlying middleware. For embedded systems with additional *para-functional* QoS constraints such as timeliness and distribution, QoS-aware component models that are aware of and provide programming and configuration mechanisms to address these constraints are needed [4].

However, even with the support of QoS-aware middleware, the process of programming and configuring both functional and para-functional properties remains tedious and error-prone. The fundamental problem is that functional and para-functional properties can *interfere* in subtle and complex ways that are often insufficiently represented, checked, or corrected in the overall system programming model. We now consider an example of interference between the inherent *functional* properties of the example application shown in Figure 1, and induced *para-functional* properties resulting from its deployment.

b. Interference Leading to Deadlock

For the example application shown in Figure 1, *functional* properties of interest are captured by a graph of invocations of facets between components. Independent of how the components are deployed on the available endsystems, the invocation graph encodes a causal ordering of the component entry points. The *para-functional* properties of interest in this example are captured by the grouping of components onto ORBs, the number of threads on each ORB, and the strategy used by each thread to wait on connections.

As we have examined in other work, using a non-blocking wait-on-connection strategy has implications for feasible schedulability of invocations, while using a blocking strategy runs a risk of deadlock [5]. We have also examined the ability of adapting rates of invocation at different points in the component packaging, assembly, and deployment lifecycle, to support resource utilization optimizations, *i.e.*, to achieve feasibility or to reduce pessimism [6].

To illustrate the more general problem of interference, we focus here on the problem of deadlock, without reference to rates, priorities, or schedulability. Specifically, we assume that an ORB thread blocks on a connection when it invokes a facet on a component on another ORB. Note that when a facet is invoked on another component in the same ORB, or even a facet within the same component, the thread of execution moves from the invoking method to the invoked method, crossing the component interface. In contrast, when a method invokes a facet on a component on another ORB, two relevant concurrency events occur:

- The thread in the method making the invocation *blocks* until the invocation completes.
- A thread in the ORB hosting the invoked facet is *bound* to the invocation.

We note that for a synchronous two-way invocation, completion does not occur until the reply message from the invoked method has been received; for an asynchronous one-way or AMI invocation, completion of the invocation occurs after the request message has been sent and any callback handlers or other post-processing mechanisms have been registered locally. With two way invocations, a case of *interference* between functional and para-functional properties emerges from this example:

- To process invocations originating from another ORB, a thread from the available ORB threads must be bound to that invocation upcall.
- If a two-way invocation of a facet on another ORB is made within the original upcall, or transitively within another method invoked within the scope of that upcall, the bound thread blocks until the remote invocation completes.
- If all threads in an ORB are blocked, no further upcalls can occur until at least one of the threads completes its original upcall and is then unbound.
- Therefore, any path in the invocation graph that *crosses into* a particular ORB more times than the number of threads in that ORB, will lead to deadlock.

Figure 2 illustrates a scenario that explores cases where deadlock does or does not occur, in a hypothetical invocation graph based on the example in Figure 1:

1. An invocation of facet s arrives at ORB 1, and binds a thread.
2. The method implementing facet s makes a blocking invocation of facet u, which binds the thread in ORB 2.
3. The method implementing facet u invokes facet v in the same thread.
4. The method implementing facet u makes a blocking invocation of facet t, which binds the second thread of ORB 1, but then returns without making further invocations which in turn unbinds the second thread in ORB 1 and unblocks the thread in ORB 2. The method implementing facet u then also returns without making further invocations, which unbinds the thread in ORB 2 and unblocks the first thread in ORB 1.
5. The method implementing facet s makes a blocking invocation of facet x, which binds the thread in ORB 3.
6. The method implementing facet x makes a blocking invocation of facet w, which binds a thread in ORB 2 – note that the same thread was bound earlier to the invocation of facet u, but was unbound upon return of the method implementing facet u.
7. The method implementing facet w makes a blocking invocation of facet y, but the only thread in ORB 3 is already blocked on the invocation from the method implementing facet x, to facet w. At this point, ORB 3 is thus deadlocked, as is the entire invocation chain through facets s, x, and w.

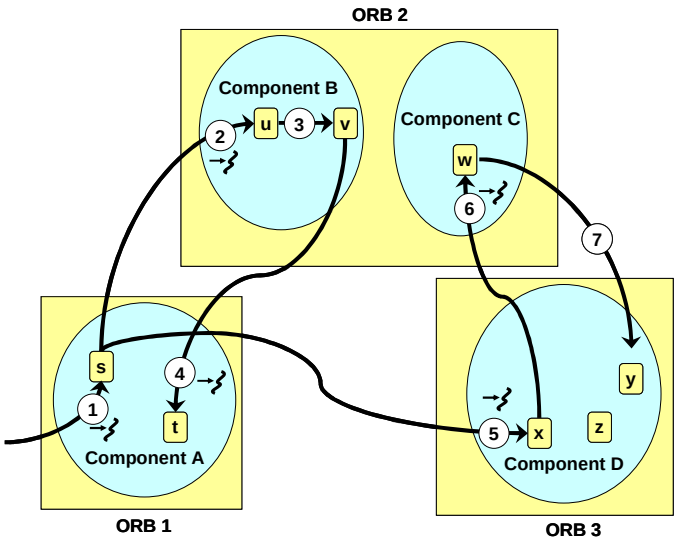

Figure 2: Interference Resulting in Deadlock

c. Toward more Complete Models of Interference

Numerous hazards other than deadlock must be avoided in complex embedded systems, such as failures of end-to-end timeliness, hardware or software faults, or even adversarial intrusion and compromise of the

system. For component-based real-time systems, temporal patterns of facet invocations and execution times of implementing methods must be modeled. The behavior of the supporting middleware must again be considered, *e.g.*, preemption policies for access to resources and the overhead from context switches and other "hidden" behaviors of the underlying middleware and operating system.

Interference due to fault-tolerance mechanisms and policies must be modeled, and measures to mitigate or repair faults must be modeled and their benefits must be balanced against their impact on other constraints, such as timeliness. For example, replication of data and invocation messages consumes computation and communication resources, but leads to more timely recovery in the event a fault occurs. A similar kind of interference between security measures and other properties is illustrated by the practice of adding watermarks to application data: computation and storage resources are consumed to increase confidence in the authenticity of information managed by the system.

C. The Problem of Appropriate Abstraction

A fundamental problem in modeling constraints in multiple dimensions such as functionality, timeliness, fault-tolerance, and security, is to provide appropriately scaled abstractions within the programming models used to develop and configure the system. Problems can arise both from excessive abstraction and from insufficient abstraction.

Excessive abstraction can result in mismatches with crucial constraints of an application. For example, if the number of threads provided by an ORB is not configurable in the programming model, hazards such as deadlock and deadline failure may result. The scenario examined in Section B.b illustrates this problem, where an application whose functional properties result in a invocation chain that crosses into an ORB more times than the number of threads it has available to perform upcalls will result in deadlock.

Changing the policy for waiting on connections to be non-blocking can avoid deadlock, but may have implications for deadline feasibility due to increased blocking times [5], and may also increase overhead by making more calls to operating system functions such as `select()`. Simply increasing the default number of threads in each ORB would alleviate the deadlock problem for some applications, but one can envision that some applications might still exhaust a given fixed limit on the upcalls a single ORB may perform due to cyclic or recursive structure in their invocation graphs. Furthermore, increasing the number of threads may increase operating system and middleware overheads, resulting in significant performance degradation.

Insufficient abstraction leads to another kind of problem, in which the details needed to configure a system are exposed in the programming model, but the space of configurations is unmanageably large, and the job of producing correct systems is thus made tedious and error-prone. This kind of *accidental complexity* has been addressed by frameworks like the ADAPTIVE Communication Environment (ACE) [7], which capture key higher level abstractions that are obfuscated by lower-level interfaces like POSIX [8].

The heterogeneity of concerns in real-world systems makes it implausible that any single configuration of properties in the supporting middleware and operating systems, when composed with the functional semantics of each application, will meet the para-functional constraints of every system. A key part of this dilemma has been called "the tyranny of the *dominant decomposition*" [9], in which a particular style of abstraction is appropriate for most of a system of constraints, but does not address others completely.

For example, ACE encodes an object-oriented decomposition of the configuration space inherent in POSIX and similar operating system interfaces such as Win32. ACE selectively uncovers certain low-level details such as file handles, which need to be shared among objects or passed to operating system interfaces, avoiding problems of either too much or too little encapsulation.

Even so, limitations of a purely object-oriented style of abstraction begin to appear when composing multiple ACE objects to create higher-level middleware subsystems such as dynamic scheduling event dispatchers [10]. In particular, information needed by different scheduling strategies may vary significantly, and using inheritance polymorphism to associate combinations of scheduling parameters with the events to be dispatched would lead to an explosion in the number of classes needed and result in an *increase* in the complexity of programming the infrastructure. Instead, applying a *generic programming* [11] style of abstraction to complement the predominantly object-oriented structure of ACE allows arbitrary scheduling parameter types to be composed with events, while preserving type safety of QoS parameters with respect to the scheduling heuristics used in the dispatcher.

Objects, distributed objects, components, and generics all treat a single type and its interfaces as the fundamental unit of encapsulation, achieving a better fit between system requirements and the abstractions available to satisfy those requirements. In real-world systems, however, key concerns still cross-cut such

single-type encapsulation boundaries leading to additional complexity in meeting their associated requirements.

The Aspect-Oriented Programming (AOP) [12] paradigm encapsulates sets of related points that cross-cut other abstraction boundaries, and thus serves to complete an appropriate set of abstractions when used in conjunction with other programming paradigms. The combination of multiple styles of abstraction avoids the problem of a single dominant decomposition, and leads to the solution approach advocated by this paper: a synthesis of model-integrated computing tools, component middleware, and lower-level aspect frameworks to organize the application and supporting infrastructure.

D. Towards Generative System Programming

This section surveys existing approaches to addressing interference between system aspects, and identifies the benefits and limitations of the current state of the art. Section D.a first describes several existing approaches to addressing interference between system aspects. Section D.b then details open research questions that these approaches do not address.

a. Survey of Current Approaches

We first survey existing approaches to managing interference between system aspects, which fall into three main categories:

- System Aspect Frameworks – system infrastructure frameworks that directly encode abstractions for managing interference.
- Model-Driven Toolsets – integrated combinations of modeling abstractions and infrastructure generation or configuration abstractions, often focused on a particular problem domain.
- Model Engineering Tools – more general abstractions for model creation, manipulation, and checking, which can be applied variously to infrastructure generation, configuration, modeling, and checking for different domains.

System Aspect Frameworks: BBN technologies *Qoskets* [13] are high-level aspect-oriented middleware abstractions for QoS management. Qoskets cross-cut distribution and layer boundaries, and thus offer end-to-end configurability of large-scale system infrastructures.

The CIAO [4] project at Washington University and Vanderbilt University extends the standard CORBA Component Model for configurability of para-functional as well as functional properties. CIAO extends the standard CCM XML-based component packaging, assembly, and deployment capabilities to include configuration of para-functional system aspects within the components, their containers, and the supporting system infrastructure. We are working with researchers at BBN to integrate their Qoskets approach with CIAO.

Researchers at the University of British Columbia have developed an AOP tool called AspectC, for C systems programming environments [14]. Complementing the AspectJ tool for Java, AspectC offers the ability to refactor many examples of open-source systems software along aspect-oriented paths. The extension of AOP tools to multiple languages is very promising and availability of such tools for C++, including the ability to weave aspects into template code, would allow new and powerful combinations of generic, object-oriented, and aspect-oriented techniques to be applied together.

Absent the availability of mature AOP tools for C++, we have tended to combine generic and object-oriented techniques with logic-driven composition approaches in the vein of work at the University of Utah on Task/Scheduler Logic [15]. In particular, we are in the process of re-factoring the Kokyu [10] dispatching framework to use both generic and object-oriented techniques in conjunction with composition logics to ensure safety of configurations with respect to para-functional properties.

The Time Weaver [1] framework developed at Carnegie-Mellon University provides system aspect configuration abstractions called *couplers* that are similar to the BBN Qoskets approach. The CMU approach explicitly promotes recursive composition of couplers for hierarchical encapsulation of abstractions, and notes several design dimensions along which para-functional aspects are composed in existing DRE systems. Of particular note in the CMU approach is the discussion of *projections* of aspects between different design dimensions, and the need to reconcile constraints, *i.e.*, to address *interference* across as well as along those design dimensions.

The CoSMIC [16] project at Vanderbilt University has similar aims to the Time Weaver project at CMU, but pursues model-driven configuration of DRE systems within the context of existing middleware frameworks, notably CIAO and QuO. By adopting both the component-oriented programming model in CIAO and the aspect-oriented programming model in the QuO

Qoskets approach, CoSMIC can leverage the appropriate style of abstraction for each of a broader set of concerns.

Model-Driven Toolsets: The Cadena [17] project at Kansas State University applies model-based techniques to configuration of component-based systems, and in particular to behavioral aspects of component-based applications and their supporting infrastructure. Cadena thus covers a wide domain of component-based applications, while allowing selective customization of the aspects relevant to each particular application. We are in the process of exploring integration of Cadena with CIAO, leveraging CIAO's ability to configure QoS properties directly within Cadena.

The VEST [18] toolkit developed at the University of Virginia is another model-driven toolset for DRE systems. Where Cadena focuses on configuring a particular kind of middleware, VEST takes a more vertically crosscutting approach, modeling, configuring and checking abstractions at the application, middleware, operating system, and even hardware levels. Both the Cadena and VEST implementations are focused on particular domains, but each is extensible to cover additional abstractions beyond those in its current implementation.

Model Engineering Tools: Whereas Cadena is focused on a particular domain, the Bogor [19] framework that is also being developed at Kansas State University provides flexible and general capabilities for developing a variety of model-checking tools. The generic modeling environment (GME) [20] tool developed at Vanderbilt University is a similarly general meta-modeling environment, and in fact was used in the VEST tool implementation.

b. Open Research Challenges for Generative System Programming

Each of the approaches surveyed above covers an important segment of the space of model-driven computing, but it is apparent that no one of those approaches can completely cover the configuration space of functional and para-functional properties in DRE systems. Therefore we argue that a synthesis of techniques is needed, to allow DRE system developers to draw on multiple tools and infrastructure frameworks and apply each to its best use, with reasonable assurance of (1) the fidelity of the abstract representations to the composed system, and (2) the correctness of the system's properties in each of its multiple design dimensions.

The following challenges must be addressed to achieve such a synthesis:

- **Implicit, ad hoc structure** of implementation frameworks must be re-factored to avoid unnecessary forms of interference, and must provide explicit reflective information for use in their composition and configuration.
- **Alternative dominant decompositions** must be supported fully in the programming model, to reduce complexity of design dimensions.
- **Higher levels of abstraction** should be used in projecting model abstractions into the implementation frameworks used to compose the model-based system.

E. AN ARCHITECTURAL VISION

This section describes an architectural vision that seeks to align the different kinds of system component infrastructures, system aspect frameworks, model-driven toolsets, and model engineering tools described in Section D.a, to address the challenges outlined in Section D.b. A key theme of this vision is that both (1) top-down modeling of application characteristics and requirements, and (2) bottom-up modeling of infrastructure aspects are necessary and appropriate. Section E.a first gives an overview of the proposed architecture. Section E.b then illustrates how segments of that architecture could serve to resolve the deadlock problem discussed in Section B.b.

a. Architecture Overview

Figure 3 illustrates the proposed architecture, which integrates *model-driven, component-oriented, and aspect-oriented* approaches.

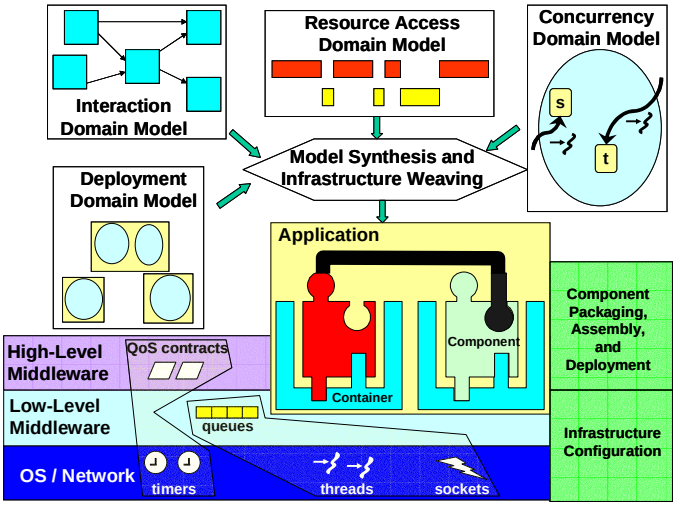

Figure 3: Model-Component-Aspect System Architecture

The following list describes the major segments of the envisioned *model-component-aspect* architecture, and the respective contribution of each segment:

- **Domain-Specific Modeling Tools** – increase scalability by encapsulating representation and analysis of individual domain models, increasing separation of modeling and checking concerns.
- **Model Synthesis Tools** – allow synthesis and checking of multiple separate domain models, a necessary adjunct to the separation of models for distinct domains.
- **Component middleware** – offers a common and standardized implementation context within which reflective information for different domain models can be applied directly to weaving and configuration of application components and supporting system infrastructure.
- **High-level system aspect frameworks** – support *integration and configuration* of existing middleware infrastructure and application implementations using their *external* interfaces.
- **Low-level system aspect frameworks** – allow *synthesis and customization* of infrastructure and components by weaving *into* those abstractions.
- **Implementation Weavers** – perform model-driven system generation, through synthesis, integration and configuration of application components and system infrastructure.

The relationships between these architectural segments are as important as the segments themselves. Application components must be able to provide model information about their para-functional constraints such as memory usage and thread safety, as well as their functional constraints, to modeling tools. Middleware infrastructure implementations must similarly provide model information about their configuration options and para-functional properties. High-level system aspect frameworks can glue together integrated configurations of existing middleware implementations [13]. Low-level system aspect frameworks are promising complements to model engineering tools, so that a domain-specific model, *e.g.*, for real-time analysis, can be co-designed with model information about specific system framework abstractions, *e.g.*, threads, timers, and queues in a dispatching subsystem [10], from which its model implementations will be generated. Component models' packaging, assembly, and deployment capabilities can be used, possibly in conjunction with domain-specific infrastructure configurators, as implementation weavers.

b. Example Resolved

We now offer a brief illustration of how various segments of a *model-component-aspect* architecture could be applied to resolve the problems illustrated in Section B.b. In the deployment scenario shown in Figure 2, the following steps could help avoid deadlock while still meeting other functional and para-functional constraints:

1. Component middleware would provide reflective information about the packaging, assembly, and deployment of components, including the invocation graph and placement of components onto ORBs.
2. Low-level system aspect frameworks would provide reflective information about default concurrency and reply-wait strategies and other properties, and offer points of configurability of those properties.
3. High-level system aspect frameworks would insert instrumentation points for collecting run-time information, such as exhaustion of a thread pool or deadline misses. Adaptation mechanisms may also be added to allow run-time manipulation of system properties, in cases where some hazards are uncommon and recovery from them is possible if encountered.
4. Domain-specific modeling tools would capture models for the system's interaction, resource access, deployment, and concurrency domains, using the reflective information supplied by the components and the aspect frameworks.
5. Model synthesis tools would integrate those models, and perform crosscutting analysis to determine ways in which the deadlock could be resolved, *e.g.*, by migrating Component C from ORB 2 to ORB 3, or by increasing the number of threads configured in ORB 3. If multiple alternatives were feasible, additional analysis would be performed to determine which would be more desirable, and under what operating conditions.
6. Finally, implementation weavers would be used to configure or even synthesize properties of the implementation – in cases where multiple configurations are possible and among them different ones would best suit different operating conditions, adaptation mechanisms would be configured to detect those conditions and adapt among the alternative configurations accordingly.

F. CONCLUSIONS AND FUTURE WORK

This paper has presented the position that interference between system aspects, in both the functional and para-functional semantics of an application, is a fundamental issue in real-world systems programming. Furthermore, it is apparent that no single dominant decomposition can address the problem of entanglement. Instead, appropriate abstractions in each of several domains must be composed and woven together in both the modeling and system generation segments of a larger integrated architecture. Model-component-aspect architectures are proposed to achieve the composition of abstractions from multiple domains, and generate DRE systems that are correct in both their functional and para-functional properties.

The challenges outlined in Section D.b offer a roadmap for future work. Of particular interest is the co-design of model engineering tools with abstraction frameworks, such as low-level aspect frameworks, that can reduce the complexity of model specification and checking, and allow more rapid development of additional domain-specific modeling tools. A further area of interest is the representations that would arise from or perhaps even precede the co-design of general model engineering tools and their abstraction libraries. In general the goal is to increase the level of abstraction within the models, and correspondingly within the implementation frameworks used to realize those models. Algebras and logics for interference-aware composition of system aspects is of particular interest, both for co-design of aspect frameworks and model engineering tools, and for customization of domain-specific models.

References

[1] R. Rajkumar and D. de Niz, “Model-Based Embedded Real-Time Software Development”, RTAS 2003 Workshop on Model Driven Embedded Systems, Washington, D.C., USA, May 2003.

[2] Object Management Group, “CORBA Components”, OMG document formal/2002-06-65, June 2002

[3] D. Alur, J. Crupi, and D. Malks, “Core J2EE Patterns: Best Practices and Design Strategies”, Prentice Hall, 2001.

[4] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall and R. E. Schantz, "Total Quality of Service Provisioning in Middleware and Applications," *Microprocessors and Microsystems, special issue on Middleware Solutions for QoS-enabled Multimedia Provisioning over the Internet*, vol. 27, no. 2, pp. 45-54, March 2003.

[5] V. Subramonian and C. Gill, “A Generative Programming Framework for Adaptive Middleware”, HICSS 2004, Hilo, HI, January 2004 (to appear).

[6] N. Wang and C. Gill, “Improving Real-Time System Configuration via a QoS-aware CORBA Component Model”, HICSS 2004, Hilo, HI, January 2004 (to appear).

[7] D. C. Schmidt, “The ADAPTIVE Communication Environment (ACE), www.cs.wustl.edu/~schmidt/ACE.html, 1997

[8] IEEE, POSIX1003.1c, “Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language]”, 1995

[9] Tarr, Harrison, Ossher, Finkelstein, Nuseibeh, and Perry, “Workshop on Multi-Dimensional Separation of Concerns in Software Engineering”, ICSE 2000, Limerick, Ireland

[10] C. Gill, D. Schmidt, and R. Cytron, “Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing”, IEEE Proceedings 91(1), Jan 2003.

[11] Matthew H. Austern, “Generic Programming and the STL: Using and Extending the C++ Standard Template Library”, Addison-Wesley, Reading, Massachusetts, 1998

[12] Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier, and Irwin, “Aspect-Oriented Programming”, Proceedings of the 11th European Conference on Object-Oriented Programming, June, 1997

[13] Schantz, Loyall, Atighetchi, and Pal, “Packaging Quality of Service Control Behaviors for Reuse”, Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing, Washington, DC, April, 2002.

[14] Coady, Kiczales, Feeley, and Smolyn, “Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code”, Joint ESEC / ACM SIGSOFT FSE-9 Conference, September 2001.

[15] Alastair Reid and John Regehr, “Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software”, 2002, citeseer.nj.nec.com/reid02taskscheduler.html

[16] Gokhale, Natarajan, Schmidt, Nechypurenko, Gray, Wang, Neema, Bapty, and Parsons, “CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications”, ACM OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA, November, 2002

[17] Hatcliff, Deng, Dwyer, Jung, and Prasad, “Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems”, Proceedings of the International Conference on Software Engineering, Portland, OR, May 2003.

[18] Stankovic, Zhu, Poornalingam, Lu, Yu, Humphrey, and Ellis, “VEST: An Aspect-based Composition Tool for Real-time Systems”, 9th IEEE Real-time Applications Symposium, Washington, DC, May 2003.

[19] Robby, M. B. Dwyer, and J. Hatcliff, “Bogor: An Extensible and Highly-Modular Model Checking Framework”, Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), to appear.

[20] Ledeczi, Bakay, Maroti, Volgysei, Nordstrom, Sprinkle, Karsai, “Composing Domain-Specific Design Environments”, IEEE Computer, November, 2001.