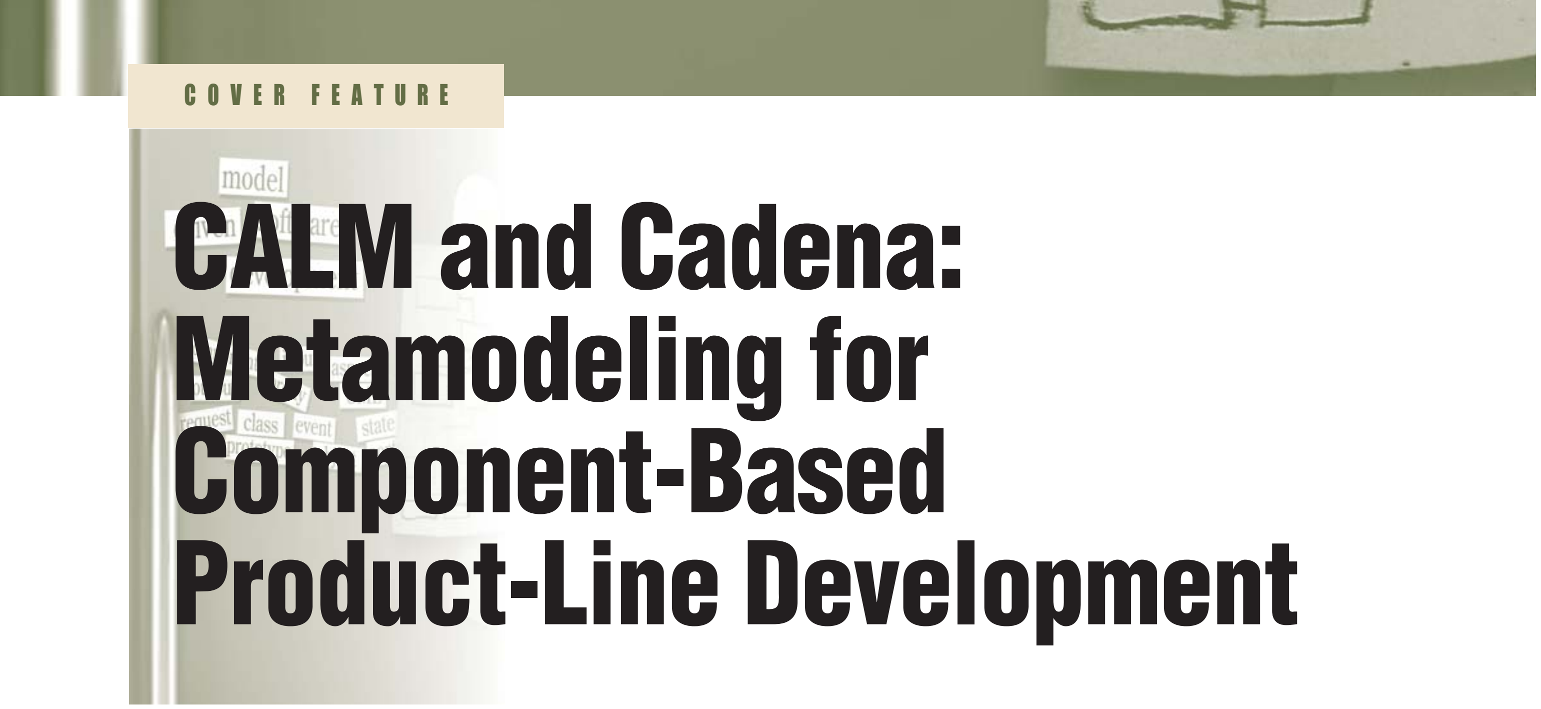


CALM and Cadena: Metamodeling for Component-Based Product-Line Development

Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff
Kansas State University

Existing software modeling approaches fail to support many product-line development activities. The Cadena platform, together with its core modeling concept, the Cadena Architecture Language with Metamodeling, addresses this deficiency by providing a highly adaptive type-centric modeling framework with robust, flexible, and extensible tool support.

Large-scale software development efforts are increasingly based on *product lines,* a development process in which developers build the software for similar product families from reusable infrastructure and common application components.[1] By emphasizing systematic reuse, the product-line approach can reduce development and production time as well as overall costs by a factor of 10 times or more.[2]

Using component middleware frameworks supports the product-line approach by

- providing well-defined interfaces that prevent client code from becoming unnecessarily tangled with low-level implementations; and
- making units easier to plug and unplug, which facilitates reuse and system evolution.

Product-line development based on such frameworks has been successful in numerous application domains ranging from large-scale distributed real-time and embedded computing systems used in mission- and safety-critical domains—including commercial air traffic control, military systems, electrical power grids, industrial process control, and medical imaging—to user-level operating-system and desktop-application integration.

A middleware-based, component-oriented software system consists of

- an integrating middleware layer that abstracts the execution environment and implements services and communication channels, and
- a network of cooperating components that run on the middleware services and enforce the business logic.

This clear separation of infrastructure and application modules, and the ability to easily compose such modules, naturally suggests three distinct development roles: The *product-line architect* sets the system architecture, selects infrastructure platforms, and organizes the development process; the *component developer* builds the business-logic modules; and the *component integrator* assembles the components into systems. The "Product-Line Development Roles" sidebar describes these roles in more detail.

Advances in architecture description languages and metamodeling environments[3] have enabled software product managers to hide the complexities of lower-level implementation details by defining structural abstractions of components, interfaces, connectors, and system assemblies that can be visualized and analyzed

Published by the IEEE Computer Society 0018-9162/06/$20.00 © 2006 IEEE

Product-Line Development Roles

Product-line architects, component developers, and component integrators each play a distinct role in middleware-based, component-oriented software product-line development.

PRODUCT-LINE ARCHITECTS

Product-line architects initially analyze commonalities and differences among systems in the product family as well as any inherent dependencies between their features and capabilities. Guided by this analysis, they construct a common software infrastructure, including a component model and supporting services, that developers can reuse for all products within the family. Product-line architects also define coding and modeling guidelines to constrain and organize the development process.

Developers can use numerous industry-standard component platforms, such as the CORBA Component Model and Enterprise JavaBeans, to support product-line development. Most out-of-the box implementations of CCM and especially EJB also add functionality that is not part of the platform specification. In addition, specialized platforms have been developed to build specific, large-scale projects—for example, the Boeing Bold Stroke/Prism model, which is used for avionics-control systems, and the Gnome-Bonobo platform, which is the basis for the Gnome desktop-manager event system.

Given the number of different component models and implementations targeted at various classes of applications (such as embedded versus nonembedded), it is often beneficial to organize the product line in a platform-independent manner by working at a level of abstraction that avoids committing to specific underlying component models or implementations. This allows using different middleware implementations for different products and can facilitate migration to middleware platforms across the product line's lifetime.

The product-line architect is responsible for developing rules, guidelines, and even automated transformations for migrating platform-independent models and artifacts to platform-specific models and implementations. In large-scale *systems of systems*, architectures may need to support multiple component models and platforms.

COMPONENT DEVELOPERS

Component developers fall into two broad categories. *Common* component developers design components that are intended to be reused across multiple projects, as well as abstract versions of more special components that are refined—for example, via inheritance—and implemented for use in specific projects.

Project-specific component developers implement components for particular systems and customize or refine common components to satisfy functional requirements for their portion of the system. Components that offer related functionality are grouped into modules or libraries.

The component developer carefully models the event and data interfaces provided to client components as well as those on which the component depends as component ports. The component integrator resolves these dependencies later during system assembly and configuration.

Components can use infrastructure services such as a persistence, time, event, or replication. Often the configuration settings for such services are exposed by the component developer in metadata associated with the component so that the component integrator can choose a particular setting after determining the component's context.

COMPONENT INTEGRATORS

Component integrators work on specific projects. They attempt to satisfy the functional and real-time requirements of a particular system by hooking together general-purpose and project-specific components drawn from a component library and by selecting distribution strategies, execution priorities, and infrastructure-specific communication services.

This phase benefits most obviously from tool support, such as automatic valuation of metadata or deployment information through implemented heuristics, or more simply through graphical display and editing of assemblies. Partial assemblies that support certain recurring functionalities (subassemblies) can be saved as libraries similar to component-type libraries.

and that can drive automatic generation of various forms of infrastructure code. However, these modeling tools are often not directly targeted to product-line development roles.

CALM/CADENA

The Cadena development-tool platform, together with its core modeling concept, the Cadena Architecture Language with Metamodeling (CALM), addresses this deficiency by providing a highly adaptive type-centric modeling framework with robust, flexible, and extensible tool support.

CALM

CALM (www.cadena.projects.cis.ksu.edu/calm) is an architecture description language that enables strongly typed modeling of platforms, components on these platforms, and component assemblies of concrete scenarios. It also supports inheritance-based hierarchical organization of platforms with aspect mechanisms for

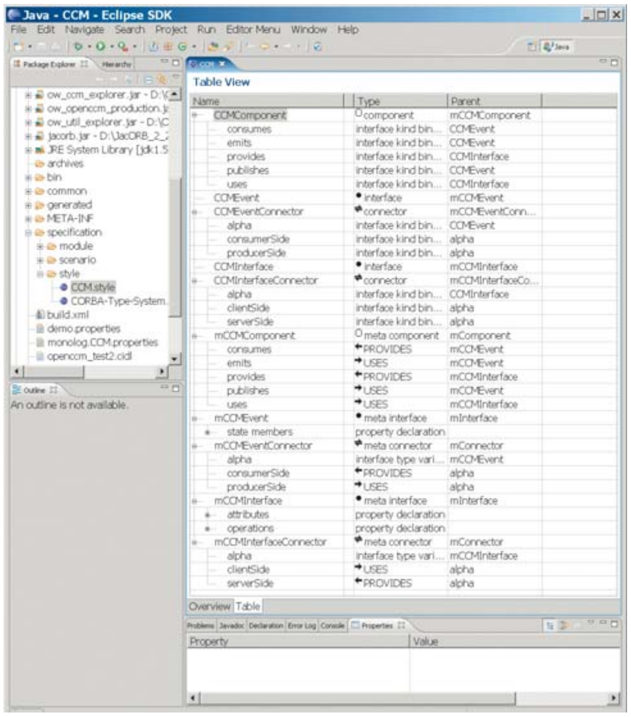


Table View

Name	Type	Parent
CCMComponent	component	mCCMComponent
consumes	interface kind bin...	CCMEvent
emits	interface kind bin...	CCMEvent
provides	interface kind bin...	CCMInterface
publishes	interface kind bin...	CCMEvent
uses	interface kind bin...	CCMInterface
CCMEvent	interface	mCCMEvent
CCMEventConnector	connector	mCCMEventConn...
alpha	interface kind bin...	CCMEvent
consumerSide	interface kind bin...	alpha
producerSide	interface kind bin...	alpha
CCMInterface	interface	mCCMInterface
CCMInterfaceConnector	connector	mCCMInterfaceCo...
alpha	interface kind bin...	CCMInterface
clientSide	interface kind bin...	alpha
serverSide	interface kind bin...	alpha
mCCMComponent	meta component	mComponent
consumes	PROVIDES	mCCMEvent
emits	USES	mCCMEvent
provides	PROVIDES	mCCMInterface
publishes	USES	mCCMEvent
uses	USES	mCCMInterface
mCCMEvent	meta interface	mInterface
state members	property declaration	
mCCMEventConnector	meta connector	mConnector
alpha	interface type vari...	mCCMEvent
consumerSide	PROVIDES	alpha
producerSide	USES	alpha
mCCMInterface	meta interface	mInterface
attributes	property declaration	
operations	property declaration	
mCCMInterfaceConnector	meta connector	mConnector
alpha	interface type vari...	mCCMInterface
clientSide	USES	alpha
serverSide	PROVIDES	alpha

Figure 1. Cadena style editor. The table, or spreadsheet, view reveals basic CORBA Component Model entities in their CALM definition.

incorporating specific platform attributes into general architectural descriptions.

The framework offers a rigorous, type-based approach for transitioning platform-independent models to platform-specific models via a series of tool-supported refinement steps. These steps incrementally incorporate more structural details regarding particular component models, underlying middleware, and product development contexts. They also augment the models with increasingly precise and detailed metadata, attributes, and weave-ins.

Basic modeling primitives in CALM are based on the three fundamental types of entities that define every component-based system and, more generally, every system of communicating processes:

- business-logic containers—the locations of computation (components);
- services—the locations of communication (connectors); and
- the interaction points between components and connectors that define the assumptions and guarantees (interfaces).

The formation and use of these entities is realized in CALM using a tiered metamodeling approach aligned with the three product-line development roles. A model in a particular tier defines the language or vocabulary of entities that can be used in constructing models in the tier below it.

Product-line architects use the top *style* tier to define architectures and capture middleware infrastructure properties by specifying languages for building types of components, interfaces, and connectors, along with appropriate metadata schemas. Component developers use those languages on the middle *module* tier to define component and interface types within a particular architecture. System integrators use the bottom *scenario* tier to allocate instances of declared component and connector types, connect allocated component instances, and configure underlying middleware, services, and network deployments by setting model-level attribute values.

While CALM exists as a textual language, developers use specialized spreadsheet-based or graphical input/output and editing environments to integrate it into Cadena, making CALM/Cadena intuitively easy to use with little training.

Cadena

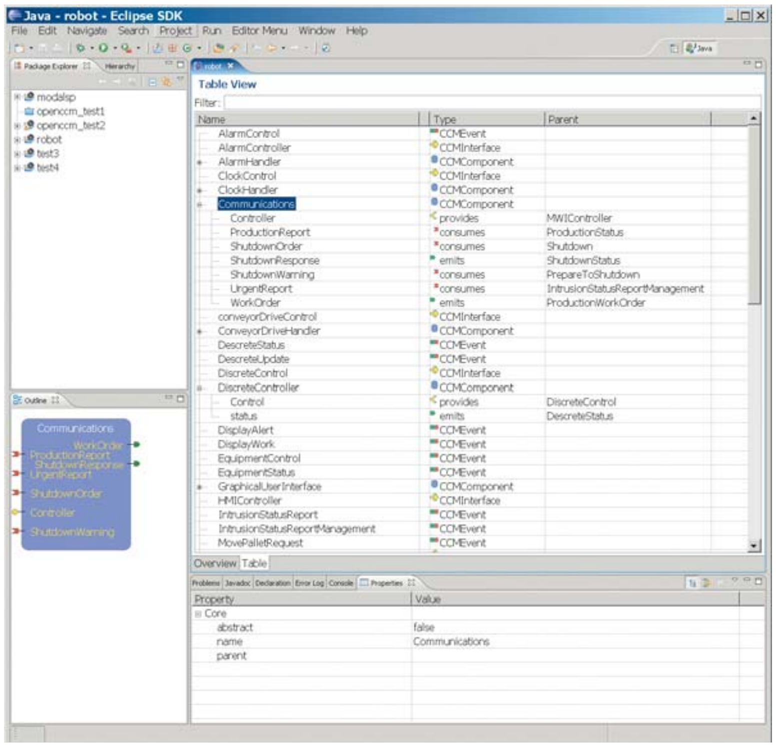
Cadena (www.cadena.projects.cis.ksu.edu) provides a variety of forms of support for creating, editing, querying, and browsing and transforming CALM models. CALM models are connected to underlying component middleware frameworks as well as analysis and code-generation facilities via Cadena plug-ins. These plug-ins effectively serve as *model interpreters* that realize the semantics of CALM models. Cadena provides a spectrum of plug-in points, letting users add specialized functionality for the component systems modeled at each CALM stage. Each plug-in is associated with a particular CALM architecture type, and inheritance on architecture types guarantees effective plug-in reuse on models as they are migrated to more specific platforms.

Cadena is engineered from the ground up to serve as an extensible tool platform rather than as a single tool. Based on Eclipse (www.eclipse.org), IBM's widely used open source integrated development environment, Cadena itself is implemented as a series of Eclipse plugs-ins. Cadena exploits the Eclipse Modeling Framework's autocoding facilities so that developers can create extremely robust, efficient tool features such as incremental type checking and model-change propagation. Cadena plug-in points are available to associate EMF adapter factories with CALM styles, enabling Cadena plug-in developers to add specialized constraint languages, constraint checkers, involved analysis frameworks, automated design heuristics, and autocoding and deployment facilities.

MODELING IN CALM/CADENA

The starting point of a CALM specification is the definition of component, connector, and interface *kinds* to describe a metamodel of a system's architectural elements at the style tier. In accordance with type theory,[4] each kind defines a language of *types* at the module tier, and every software code unit, or *instance,* at the scenario tier is derived from a type. For example, a single component instance has a certain component type, and the component type is defined within its component kind. To specify component kinds, CALM uses the concept of *metakinds*—toolboxes that serve as a factory for new types of components, interfaces, and connectors.

The relation between instance and type is the same as that between type and kind. Although the concept is simple, this additional level of abstraction gives CALM/Cadena the flexibility to provide specialized development environments for a huge class of platforms.



Java - robot - Eclipse SDK

Table View

Name	Type	Parent
AlarmControl	CCMEvent	
AlarmController	CCMInterface	
AlarmHandler	CCMComponent	
ClockControl	CCMInterface	
ClockHandler	CCMComponent	
Communications	CCMComponent	
Controller	provides	MWIController
ProductionReport	consumes	ProductionStatus
ShutdownOrder	consumes	Shutdown
ShutdownResponse	emits	ShutdownStatus
ShutdownWarning	consumes	PrepareToShutdown
UrgentReport	consumes	IntrusionStatusReportManagement
WorkOrder	emits	ProductionWorkOrder
conveyorDriveControl	CCMInterface	
ConveyorDriveHandler	CCMComponent	
DescreteStatus	CCMEvent	
DescreteUpdate	CCMEvent	
DiscreteControl	CCMInterface	
DiscreteController	CCMComponent	
Control	provides	DiscreteControl
status	emits	DescreteStatus
DisplayAlert	CCMEvent	
DisplayWork	CCMEvent	
EquipmentControl	CCMEvent	
EquipmentStatus	CCMEvent	
GraphicalUserInterface	CCMComponent	
HMIController	CCMInterface	
IntrusionStatusReport	CCMEvent	
IntrusionStatusReportManagement	CCMEvent	
MovePalletRequest	CCMEvent	

Property	Value
Core	
abstract	false
name	Communications
parent	

Figure 2. Cadena module editor. While operating under the CALM CCM style, the editor adapts to provide customized CCM support.

Style tier

Figure 1 shows the Cadena style editor with the Common Object Request Broker Architecture (CORBA) Component Model specification (www.omg.org/technology/documents/formal/components.htm). The table, or spreadsheet, view reveals basic CCM entities in their CALM definition, such as the highlighted CCMComponent with its capabilities to send and receive events (consumes, emits, publishes) or to provide and use data interfaces (provides, uses). Note that the CCMComponent is derived from a specific template, the metakind mCCMComponent.

Component kinds. These describe a platform's software unit primitives and possible mechanisms that these units may use to interact with one another. The possible interaction points, called *port options* in CALM, provide a means to declare explicit context dependencies—a central theme of component-based development. The product-line architect uses port options to specify the kinds of interfaces a component kind can feature, together with the necessary keywords—for example, *consumes* for CCM event input ports, and *publishes* or *emits* for CCM event output ports, as shown in Figure 1.

Further, developers can add *attributes,* typed through CALM's own extensible type system, to the component kind to define its ability to carry metadata. These attributes—which can be specified for the whole kind, types therein, or concrete instances—can hold data for deployment information, initialization values, and all sorts of functional, structural, or organizational content.

Connector kinds. These model the services the platform provides. The chosen platform can include various middleware services supporting intercomponent communication, distribution, persistence, and state replication, among others. Each connector kind represents a distinct platform service or family of services. Each connector definition may contain multiple role declarations, much like port options in component kinds, and interface-type constraints governing each role. Single-role connectors abstract services such as time-out generators, while multirole connectors model intercomponent communication services.

Note that connectors are abstractions, which are not meant as descriptions of the services' implementations. Rather, they describe the service's intended usage and thus enforce certain coding styles.

Interface kinds. These categorize interaction points of platform components and check component-connector compatibility. For example, the CCM component interacts with the asynchronous connector through event interfaces and the synchronous connector through CCM data interfaces. Interface kinds are defined by a set of attributes in the same manner as component kinds. For the CALM CCM style, for example, module-tier attributes for interface kinds describe possible method signatures of CCM interface types.

After an architectural style is specified within Cadena, the tool suite does not remain static. Rather, upon alteration of a style, the lower tiers adapt and offer new and changed primitives in that style's language.

Module tier

Working with the kinds—that is, type languages—defined for a particular architectural style, component developers use the module tier to define libraries of design-time component types that conform to that style. Component and interface kinds from the style are available in the Cadena module editor, shown in Figure 2, to

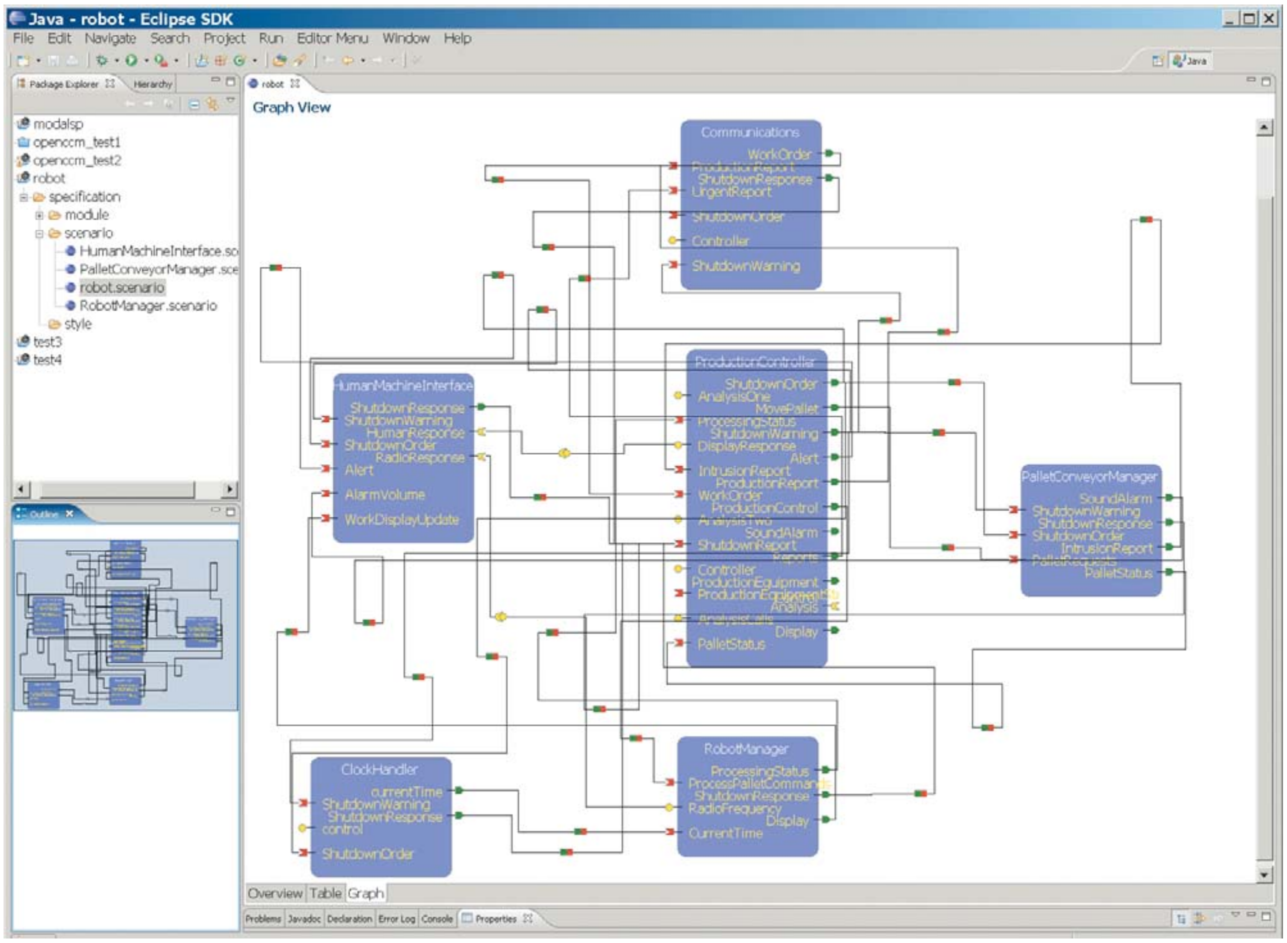


Figure 3. Cadena scenario editor. The user can insert new components and configure connections through drag-and-drop matching.

create component and interface types. The library of design-time types in the module is then available for allocating and connecting component instances at the scenario tier.

Building types at the module tier is analogous to defining component types using the CCM Interface Definition Language in the standard CCM development process. Module-tier component types are typically associated with a Cadena autocoding plug-in that generates stubs, skeletons, and other infrastructure code required for component implementation. After applying autocoding plug-ins to generate skeleton implementations, component developers can use Eclipse development environments—for Java or C++, for example—to complete component implementations by coding the business logic associated with components.

Scenario tier

To allocate and connect component instances at the scenario tier, Cadena offers two standard development environments: a tree-based editor and a graphical editor. Figure 3 shows the graphical editor open to a CCM system. At the top is an instance diagram representing the Communication component shown in Figure 2. Within the graphical editor, the user can insert new components and configure connections through drag-and-drop matching of the connector role and component port.

Large system assemblies with many components can overwhelm developers and inhibit their ability to reason effectively about a system. To help developers build important structural abstractions that collect multiple cooperating components together in a single unit, Cadena supports the notion of a *nested assembly*—essentially a virtual component that is actually realized as a nested network of connected component instances. Cadena's graphical scenario editor allows a developer to form a nested assembly from a set of components, view this nested assembly as one entity in the graphical scenario editor, and expand and edit the nested assembly as needed.

PRODUCT-LINE SUPPORT

Product-line development includes many involved tasks such as designing and establishing platform specifications, refining these specifications for more specific projects, combining distinct platform specifications into hybrid platforms, migrating components and assemblies from one platform to the next, augmenting a platform specification to accommodate deployment or metadata,

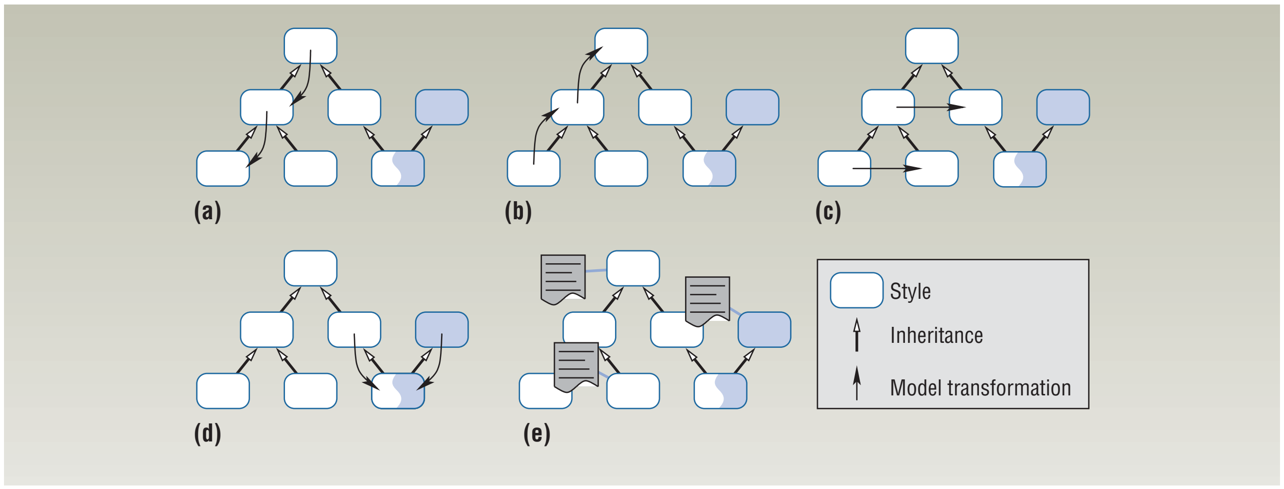


Figure 4. CALM style hierarchies. (a) Model specialization. (b) Model abstraction. (c) Model migration. (d) Hybrid model construction. (e) Attribute sheet attachment.

and integrating standard functionality into different platforms.

Inheritance hierarchies

To support these tasks, CALM introduces *inheritance hierarchies,* an object-oriented programming concept, into style design. For one CALM style specification to inherit from another means that the component, connector, and interface kinds defined in the parent style are also present in the substyle. CALM further allows *multiple inheritance*—that is, a style can name multiple parent styles.

In addition to facilitating style specifications, a well-organized hierarchy of architectural styles serves as a conceptual basis for code reuse over distinct platforms by providing a guide for type-safe migration of artifacts from compliance with one style to compliance with another.

As Figure 4a shows, the specialization of generic platform specifications toward more individual platforms corresponds to a parent-to-child transition, while the abstraction from a particular platform to a more generic, reusable form corresponds to a child-to-parent transition, as Figure 4b shows.

In addition, the reuse of elements from one platform on another generally corresponds to model migration, as Figure 4c illustrates. Finally, as Figure 4d shows, one platform inheriting from multiple platforms yields a hybrid or combination platform. Developers also can use this operation to establish integrative styles that mitigate between platforms as well as to factor in standard definitions that are reusable over multiple, otherwise unrelated styles.

Attribute attachment

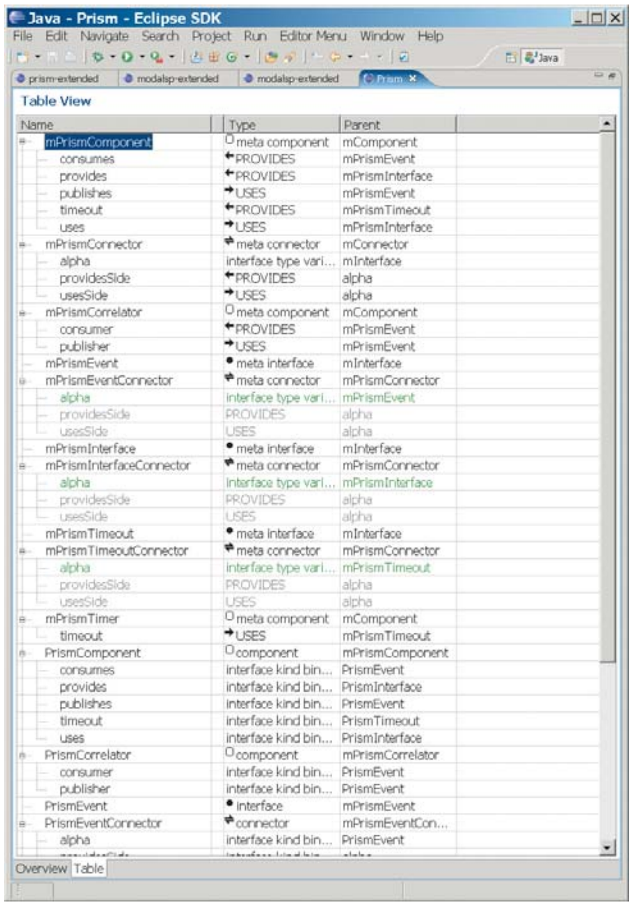
A second mechanism for refining architectural styles, and models therein, is *attribute attachment,* shown in Figure 4e. CALM offers an extendable value-type system that allows the definition of new types through names or combinations of existing types. Attributes typed in this system can be defined directly within a style specification or on separate attribute sheets, which can be added at every model tier.

Some values that attributes capture can be specific to particular scenarios, such as deployment or location information. Other sorts of attributes recur in a general context, such as structures used to describe method signatures in interfaces, or in specific contexts such as component run rates in real-time systems. Developers can use attribute sheets to factor in general data over multiple systems as well as to factor out specific data in particular systems—for example, to allow viewing a projected system's structural interrelations without having to set real-time data.

EXAMPLE PRODUCT LINE

One of the benchmark test cases for CALM/Cadena is Boeing's Avionics Open Experimental Platform (OEP), developed as part of the Defense Advanced Research Projects Agency's Program Composition for Embedded Systems project and available for exchange within the academic research community. The Avionics OEP is inspired by Boeing's Bold Stroke mission-control software product-line architecture for its military avionics platforms such as the F-18.

The OEP implements Prism, a model that consists of three component kinds. The *business component* is equivalent to the CCM component—that is, it is the generic container for business logic. The *correlator* is a specialized component that lowers network load through asynchronous message filtering. The *event-channel component* serves as the source for periodic time-out events. The event-channel component models an infrastructure unit that also manages threading and buffers asynchronous communication, but these tasks are deliberately hidden in the abstraction of the services.


Java - Prism - Eclipse SDK
File Edit Navigate Search Project Run Editor Menu Window Help
prism-extended
modalsp-extended
modalsp-extended
Prism
Table View
Overview
Table

Name	Type	Parent
mPrismComponent	meta component	mComponent
consumes	PROVIDES	mPrismEvent
provides	PROVIDES	mPrismInterface
publishes	USES	mPrismEvent
timeout	PROVIDES	mPrismTimeout
uses	USES	mPrismInterface
mPrismConnector	meta connector	mConnector
alpha	interface type vari…	mInterface
providesSide	PROVIDES	alpha
usesSide	USES	alpha
mPrismCorrelator	meta component	mComponent
consumer	PROVIDES	mPrismEvent
publisher	USES	mPrismEvent
mPrismEvent	meta interface	mInterface
mPrismEventConnector	meta connector	mPrismConnector
alpha	interface type vari…	mPrismEvent
providesSide	PROVIDES	alpha
usesSide	USES	alpha
mPrismInterface	meta interface	mInterface
mPrismInterfaceConnector	meta connector	mPrismConnector
alpha	interface type vari…	mPrismInterface
providesSide	PROVIDES	alpha
usesSide	USES	alpha
mPrismTimeout	meta interface	mInterface
mPrismTimeoutConnector	meta connector	mPrismConnector
alpha	interface type vari…	mPrismTimeout
providesSide	PROVIDES	alpha
usesSide	USES	alpha
mPrismTimer	meta component	mComponent
timeout	USES	mPrismTimeout
PrismComponent	component	mPrismComponent
consumes	interface kind bin…	PrismEvent
provides	interface kind bin…	PrismInterface
publishes	interface kind bin…	PrismEvent
timeout	interface kind bin…	PrismTimeout
uses	interface kind bin…	PrismInterface
PrismCorrelator	component	mPrismCorrelator
consumer	interface kind bin…	PrismEvent
publisher	interface kind bin…	PrismEvent
PrismEvent	interface	mPrismEvent
PrismEventConnector	connector	mPrismEventCon…
alpha	interface kind bin…	PrismEvent

Figure 5. Boeing Bold Stroke/Prism style in Cadena. Prism offers an untyped, asynchronous, event-propagation mechanism with a statically set publish-subscribe connection scheme as well as a synchronous, typed, data transmission service.

Figure 5 shows a table view of the Prism style in Cadena. Prism offers two types of services: an untyped, asynchronous, event-propagation mechanism with a statically set publish-subscribe connection scheme (event connection) and a synchronous, typed, data transmission service (interface connection).

Figure 6a illustrates using Cadena to model a single-processor modal scenario from the OEP. As Figure 6b shows, Cadena seamlessly supports the three different kinds of Prism components, which can be assigned custom shapes, within the three CALM tiers.

Adding to the communication infrastructure

Like many other Prism examples, ModalSP exploits *control-push data-pull*, a dual connection in which a component supplying data first sends an asynchronous message announcing that new data is available, then the receiving component uses a synchronous connection to retrieve the data. This guarantees that the synchronous connection never blocks or waits for updates. The control-push data-pull strategy links many components by two connectors instead of one. When developers use this approach, tool support must be available to ensure that both connectors' end points match. In CALM, it is also possible to define a dedicated connector with both synchronous and asynchronous connection points that captures the control-push data-pull protocol in a single entity.

Adding real-time metadata

In addition to the communication network, a Prism system configuration consists of deployment information and metadata such as initialization values, security levels, run rates, and location data. CALM features an attribute-attachment system that lets users weave in attributes orthogonal to the style hierarchy on each tier of the development process. Cadena provides multiple forms of support for attribute management, including declaration and checking of domain-specific attribute types, and plug-in facilities for processing attribute data values.

For example, Prism developers must assign a priority value to each component event-handler port following a particular set of development heuristics that involve tracing data, event, and control dependencies across networks of components. In the Cadena Prism development environment, a Prism plug-in automates calculation of these priority settings by coding the previously manually implemented heuristics, significantly reducing the time and effort required for this particular task.

This is an instance of a general strategy for using a model-centric approach to minimize development effort and increase overall confidence levels: Important system attributes are presented at the modeling level and various domain-specific analyses, configuration mechanisms, automated developer advice, and integrity checks are automated through plug-ins tailored to a particular development context.

Advanced platform organization

The CALM style hierarchy is more than a mechanism for capturing features or abstractions of existing component platforms; it provides a structured approach for leveraging commonalities and capturing differences across families of closely related platforms. For example, the Boeing Bold Stroke line entails some abstract parent styles from which the actual Prism platform definition inherits, as well as child styles for particular platforms such as the F/A-18 Hornet fighter, which in turn branches into the F/A-18E and F/A-18F Super Hornet aircraft.

The guiding principle for style inheritance is the reuse of infrastructure implementation as represented by CALM kinds. Reimplementations and new additions within the substyle, which appear in the definition of the substyle as new kinds, integrate naturally into the specification.

Systems designed for distinct platforms often must cooperate in personal computing—for example, desk-

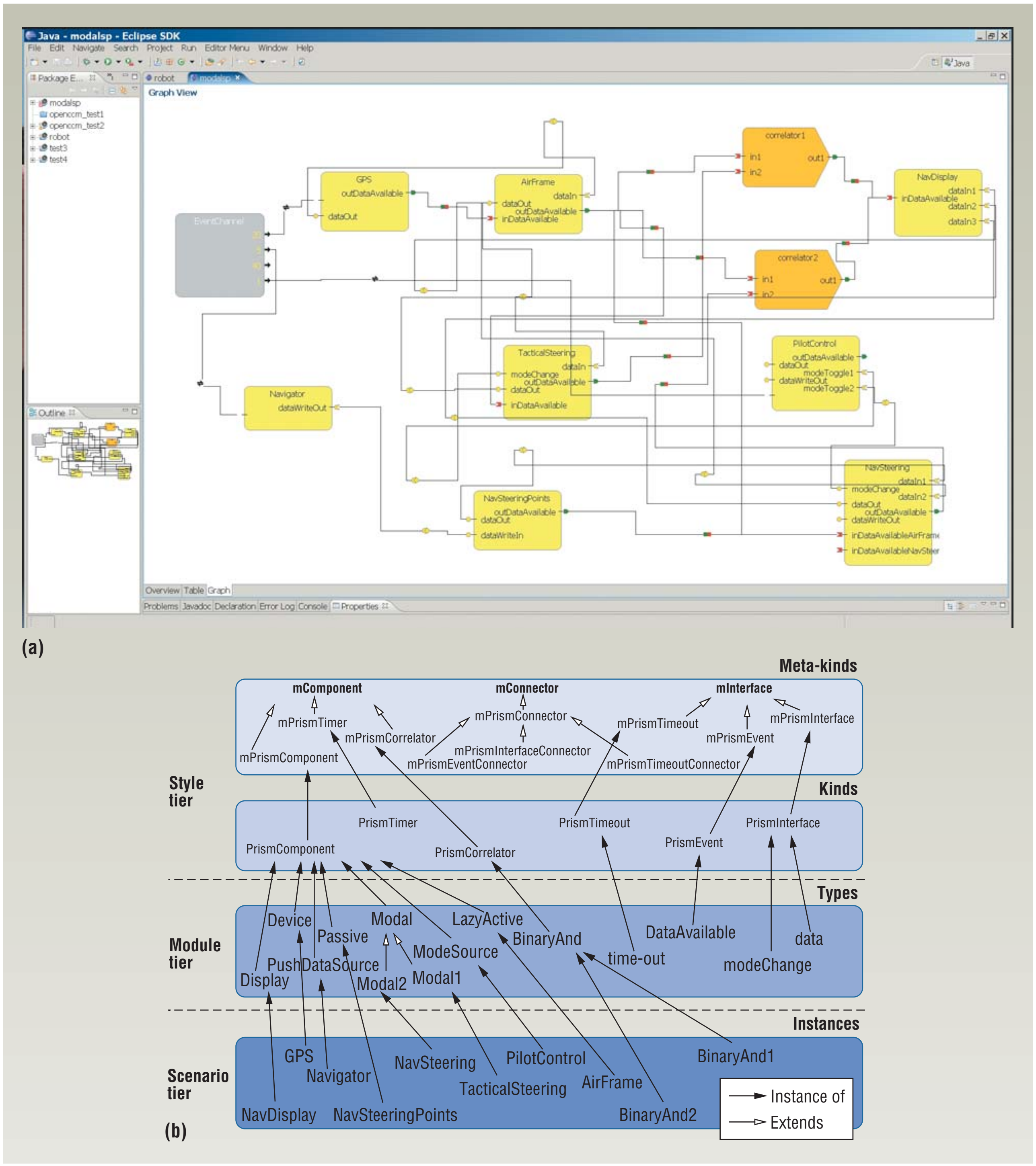


Figure 6. Boeing's single-processor modal scenario in Cadena. (a) Graphic view. (b) Specifying ModalSP within CALM.

top systems integrate CCM and Bonobo components—and even more frequently in companywide business integration. CALM is the first conceptual framework to accomplish straightforward type-safe merging of platforms. The strategy is to create a child style that inherits from each style to be merged. Within the child style, developers can easily specify necessary components or services that act as mitigators between the parent styles.

The design principles underlying CALM/Cadena have been influenced by extensive interaction with product-line architects at Boeing Phantom Works and Lockheed Martin. In addition, collaboration with real-time middleware and modeling experts at Vanderbilt University to integrate Cadena with the school's CoSMIC (Component Synthesis with Model-Integrated Computing) CCM environment (www.dre.vanderbilt.edu/

cosmic), Generic Modeling Environment (www.isis.vanderbilt.edu/Projects/gme), and CIAO (Component Integrated ACE ORB) real-time component middleware (www.cs.wustl.edu/~schmidt/CIAO.html) have provided significant opportunities to enhance and refine our product-line modeling ideas.

Compared to many existing approaches that focus on modeling and analyzing single-system assemblies in a fixed component model, CALM/Cadena is a rigorous type-based framework for modeling multiple component middleware platforms, systematically organizing and transitioning between platform definitions, and creating customized development environments that leverage domain knowledge and automate development process steps to enable early design decisions for entire product lines.

The framework continues to evolve. The initial set of features in Cadena 2.0, soon to be released, focuses on capturing and specifying basic entities with their structural interrelations. This will enable users to establish platform terminologies, build and assess topologies, provide basic autocoding and deployment facilities, design attribute-seeding mechanisms, and conduct fundamental analysis such as slicing and cycle detection. Because a main focus of Cadena is its extensibility, custom views of component systems can also easily be added.

As for CALM, research is under way to flexibly attach various forms of behavioral specification. This will enable much finer grained dependence analysis and evaluation of systems' temporal behavior, both absolute timing (real-time and schedulability analysis) and relative timing (temporal logic properties, model checking, and safety analysis).

More information about CALM/Cadena and other analysis and verification tools built by the Laboratory for Specification, Analysis, and Transformation of Software at Kansas State University can be found at the SAnToS Lab home page (www.cis.ksu.edu/santos). ■

Acknowledgment

This work was supported in part by the Defense Advanced Research Projects Agency as part of the Program Composition for Embedded Systems project (contract F33615-00-C-3044) and the Army Research Office's High-Confidence Embedded Systems initiative (contract DAAD19-01-1-0564).

References

1. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns,* Addison-Wesley Professional, 2002.
2. K. Schmid and M. Verlage, "The Economic Impact of Product Line Adoption and Evolution," *IEEE Software,* vol. 19, no. 4, 2002, pp. 50-57.
3. A. Ledeczi et al., "The Generic Modeling Environment," *Proc. IEEE Int'l Workshop Intelligent Signal Processing,* 2001; www.isis.vanderbilt.edu/publications/archive/Ledeczi_A_5_17_2001_The_Generi.pdf.
4. B.C. Pierce, *Types and Programming Languages,* MIT Press, 2002.

***Adam Childs** is pursuing an MS in computer science at Kansas State University. His research interests include software architecture, model-driven development, and component-based software. He is a member of the ACM. Contact him at achilds@cis.ksu.edu.*

***Jesse Greenwald** is a research associate in the Department of Computing and Information Sciences at Kansas State University. His research interests include model-driven development and component-based software. Greenwald received an MS in software engineering from Kansas State University. Contact him at jesse@cis.ksu.edu.*

***Georg Jung** is a PhD candidate and graduate research assistant in the Department of Computing and Information Sciences at Kansas State University. His research interests include specification, typing, and verification of software architectures and component-based systems. Jung received a Diplom in computer science from Saarland University, Germany. Contact him at jung@cis.ksu.edu.*

***Matthew Hoosier** is a research associate in the Department of Computing and Information Sciences at Kansas State University. His research interests include model-checking and software specification and verification techniques. Hoosier received an MS in computer science from Kansas State University. Contact him at matt@cis.ksu.edu.*

***John Hatcliff** is a professor in the Department of Computing and Information Sciences at Kansas State University, where he also leads the SAnToS Laboratory. His research interests include model-driven development, program analysis and verification, and software engineering for safety- and mission-critical systems. Hatcliff received a PhD in computer science from Kansas State University. He is a member of the ACM. Contact him at hatcliff@cis.ksu.edu.*