

AFRL-VA-WP-TR-2006-3048

**TECHNOLOGIES, DEVELOPMENT
TOOLS, AND PATTERNS FOR
AUTOMATIC GENERATION AND
CUSTOMIZATION OF ADAPTABLE
DISTRIBUTED REAL-TIME AND
EMBEDDED (DRE) MIDDLEWARE**

**Dr. John Hatcliff, Dr. Matthew Dwyer, Dr. Masaaki Mizuno,
Dr. Gurdip Singh, and Gary Daugherty**

**Kansas State University
Sponsored Projects Accounting Controller's Office
10 Anderson Hall
Manhattan, KS 66506**



DECEMBER 2005

Final Report for 31 March 2003 – 30 March 2005

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

**AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542**

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site (AFRL/WS) Public Affairs Office (PAO) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS 06-0767, 21 Mar 2006.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

//S//

Vincent W. Crum, Program Manager
Control Systems Development
and Applications Branch

//S//

Michael P. Camden, Chief
Control Systems Development
and Applications Branch

//S//

Brian W. Van Vliet, Chief
Control Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YY) December 2005		2. REPORT TYPE Final		3. DATES COVERED (From - To) 03/31/2003– 03/30/2005		
4. TITLE AND SUBTITLE TECHNOLOGIES, DEVELOPMENT TOOLS, AND PATTERNS FOR AUTOMATIC GENERATION AND CUSTOMIZATION OF ADAPTABLE DISTRIBUTED REAL-TIME AND EMBEDDED (DRE) MIDDLEWARE				5a. CONTRACT NUMBER F33615-00-C-3044		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER 0602301		
6. AUTHOR(S) Dr. John Hatcliff, Dr. Matthew Dwyer, Dr. Masaaki Mizuno, Dr. Gurdip Singh, and Gary Daugherty				5d. PROJECT NUMBER A04I		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER 0B		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kansas State University Sponsored Projects Accounting Controller's Office 10 Anderson Hall Manhattan, KS 66506				8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Vehicles Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson Air Force Base, OH 45433-7542				10. SPONSORING/MONITORING AGENCY ACRONYM(S) AFRL-VA-WP		
				11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-VA-WP-TR-2006-3048		
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES Report contains color. PAO Case Number: AFRL/WS 06-0767, 21 Mar 2006.						
14. ABSTRACT Large-scale, distributed real-time and embedded (DRE) systems are increasingly being used to control critical aspects of DoD systems. PCES work has shown how model-integrated computing and adaptive and flexible middleware frameworks can be applied for defining, analyzing, generating, and customizing large-scale high-assurance, high-performance DRE systems. KSU aims to provide development frameworks that contain as a centerpiece a variety of forms of software models. We have been able to demonstrate that extending software modeling tools with analysis and optimization tools tied to actual development processes can dramatically decrease costs associated with developing DRE systems. To validate the technologies we developed, we have built a model-integrated development environment called Cadena. Cadena provides a variety of capabilities for model-driven implementation and analysis of component middleware systems. Through the PCES project, we have demonstrated that Cadena can dramatically reduce the effort required to construct component-based systems in the context of product-line architectures such as Boeing's Bold Stroke avionics mission-control software. Moreover, Cadena's verification and "correct-by-construction" modeling techniques provide increased confidence in the safety and correctness of the resulting system.						
15. SUBJECT TERMS Model-driven development, middleware systems for real-time, model analysis, automatic customization, Distributed embedded systems, aspects, prescriptive aspects, software composition tools, software analysis tools, component based design						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT: SAR	18. NUMBER OF PAGES 52	19a. NAME OF RESPONSIBLE PERSON (Monitor) Vincent W. Crum	
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (Include Area Code) N/A	

1 Problem Description and Project Overview

1.1 Research Challenges Driving PCES Research

Large-scale, distributed real-time and embedded (DRE) systems are increasingly being used to control critical aspects of our nation's infrastructure. For instance, DRE systems are now deployed in commercial air traffic control, military systems, electrical power grid, and industrial process control, and medical imaging domains. Numerous sectors of our government and economy depend on the stability, security, and robustness of these systems.

In the context of military systems, to accomplish its Joint 2010 and Joint 2020 Operation Visions the DoD is seeking next generation combat vehicles and systems that provide:

- Revolutionary network-centric capabilities via collaboration and integration of hundreds of disparate extent and to-be-developed systems.
- Enhanced automation and software control of ground, air, sea, and space assets with long-range precision capabilities and assured, sustainable mission performance.
- Affordable support costs over a 20 to 40 year life-cycle that will include numerous technology refresh and capability upgrades.

Highly-reliable DRE systems will be necessary to support emerging national infrastructure as summarized above as well as the operational platforms required for achieving the DoD's objectives of network-centric collaboration of systems/platforms, highly-autonomous systems, and distributed and evolving command-and-control structures. It is tedious, error-prone, and costly to develop, optimize, validate, and deploy these types of DRE systems using conventional software technologies. Once developed, such systems need to be evolvable so that they can be easily refreshed with new technologies (*e.g.*, incorporation of new banks of sensors with increased precision, enhanced displays, mission control systems, etc.), and system components need to be transitioned for use in successive similar but distinct platforms (*e.g.*, the F/A-18A,B,C,D,E,F aircraft platforms) and missions (*e.g.*, insertion of updated communication devices and protocols for joint operations with NATO allies).

Regardless of the particular domain, large-scale DRE systems share the following characteristics that make them extraordinarily complex and very hard to design, implement, validate/certify, maintain, and evolve:

- *Systems of systems.* Large-scale DRE systems are often organized as "systems of systems" where functional and QoS requirements must be satisfied using the capabilities of many subsystems that were often not designed for integration or interoperability.
- *Large-scale, complex, network-centric operations.* The scope of large-scale DRE systems requires the applications they host to be distributed over networks, rather than being deployed as isolated stand-alone entities. Moreover, the complexity of these systems makes it hard for integrators to have a complete and coherent view of their requirements and implementations. Likewise, developers of particular modules who should be insulated from other development aspects are often overwhelmed with artifacts that are irrelevant for their specific tasks.
- *Heterogeneous platforms.* As they are distributed across networks, DRE systems are often deployed on a variety of computing platforms that are interconnected by different types of networking technologies with varying levels of QoS. This heterogeneity makes it hard for applications running on DRE systems to meet their end-to-end functional and QoS requirements in a systematic and predictable manner.
- *Incorporating legacy systems and technology refresh.* Construction of systems of systems often entails integrating legacy systems that were not designed for interoperability. Moreover, because DRE are often required to be long-lived, functional aspects of the system often need to be "refreshed" (*i.e.*, replaced with corresponding pieces that represent updated technology), but often functionalities to be replaced are "tangled" with the rest of the system so that cannot be easily decoupled and swapped out.
- *Need for families of systems.* DoD platforms usually evolve through multiple versions over time, to support technology updates, customized versions for sale to allies (*e.g.*, the F-16I – Israeli version of the F-16), trainer versions (*e.g.*, the two-seater F/A-18B). Though these different versions have many commonalities, current development practices make it difficult to directly reuse software across platforms since code is often

tied directly to particular hardware, resulting in costly maintenance of multiple versions of code with similar functionality.

1.2 Overview of Technologies Emphasized In PCES

PCES researchers have sought to address the challenges above using a variety of technologies including *component middleware*, *domain specific modeling techniques*, and *various forms of flexible model and code level analyses*.

1.2.1 Component Middleware

Over the past decade researchers have sought to develop, optimize, and standardize *object-oriented middleware* [HV99]. Middleware is systems software that (1) resides between applications and the underlying operating systems, network protocol stacks, and hardware and (2) helps shield DRE applications from having to custom-build interfaces to differing platforms and from having to develop low-level code needed to transform data into a form that can be moved across network connections. Middleware is often described as the “glue code” or “plumbing” that hooks multiple applications together and routes data and information transparently between different back-end data sources. Object-oriented middleware also implements reusable *services* (such as asynchronous event-based communication, global scheduling, and dynamic resource management) that provide functionality common to many DRE applications. Together, the use of middleware and supporting services can dramatically reduce the amount of effort required to build and maintain DRE applications. Examples of object-oriented middleware for DRE systems include Real-time Java [BGB⁺00] run-time environments (*e.g.*, jRate [Ang03]) and Real-time CORBA object request brokers (ORBs) [Obj02] (*e.g.*, TAO [SLM98]).

More recently, *component middleware* [HC01] has defined additional capabilities that enhance object-oriented middleware for DRE systems, as follows:

- It defines a component abstraction that consists of a collection of (1) *interface ports*, exposing operations clients can invoke to use a service provided by the component, or indicating the methods/services that the component itself depends on to achieve its functionality and (2) *event ports* that are used to publish and consume events. These facilities provide well-defined interfaces that hide implementations (keeping client code from becoming unnecessarily tangled with low-level implementations in the component) and make units easier to plug and unplug.
- It allows developers to focus on programming their application “business logic” (*i.e.*, primary functionality), rather than wrestling with lower-level tasks (*e.g.*, network programming, scheduling, security, and event processing). Instead, the application functionality is associated with the configuration-related capabilities via auto-generated component “glue code” that standardizes the interaction with other components and the middleware.
- It supports component *containers* that define a common operating environment in which a set of related components execute. Containers also provide components with key resources (*e.g.*, priority levels, real-time threads of control, and transparent state replication) and shield components from many tedious, error-prone, and non-portable complexities of the underlying networks, operating systems, and object-oriented middleware.
- It makes a significant attempt to address problems of configuring and deploying DRE applications throughout networks of heterogeneous computing nodes by providing standardized component assembly, packaging, and distribution formats. These configuration and deployment mechanisms enable the core functional issues to be decoupled from QoS-related issues so that QoS properties can be developed, configured, monitored, and managed not by those developing application functionality, but by a separate set of specialists (*e.g.*, middleware developers, systems engineers, and administrators) who are often much better positioned to make the appropriate configuration and deployment choices.

Examples of component middleware for DRE systems includes Prism [SR03] and the Lightweight CORBA Component Model (CCM) [Obj03] (*e.g.*, CIAO [WSG⁺03]).

Despite being in its infancy, component middleware is already having a positive impact on the quality and productivity of developing DRE systems, such as distributed interactive simulations [Nos02], avionics mission

computing systems [SR03], and distributed multimedia applications [LGG⁺01]. By defining additional capabilities and common services – as well as standardizing configuration and deployment mechanisms – component middleware enables DRE application developers to concentrate on the application-specific aspects of their systems, and leave the application-independent communication, deployment, and QoS-related details to DRE middleware developers. As a result, an increasing number of DRE systems are being developed using component middleware.

1.2.2 Model-driven Development Techniques

The large scale and dynamic nature of next-generation DRE systems will make development testing, maintenance, and technology refresh increasingly hard. Model-based software development is a key emerging technology for next-generation DRE systems [HG96, Lin99, SK97, Obj01a]. Software models provide abstract representations of software structure and behavior that allow developers and analysis tools to focus only on the software features exposed in the model without being overwhelmed by a myriad of low-level details.

Conventional modeling environments have proven useful by providing graphical views of system aspects, but these views are often no more than boxes and arrows that represent static structure while omitting deeper semantic functional and QoS properties, information about adaptability, and control of specialized entities. Moreover, while many development environments support class/object-based modeling of systems [Mat99, Obj01a, Obj01b], few environments support modeling for component-based system development tasks, such as component interface definition, component assembly, and component deployment, and fewer still provide the additional QoS management needed in the DRE domain.

In our PCES work, our team sought to emphasize that modeling is not only central in organizing the system's structure, it is also the vehicle by which analysis results are generated and displayed, and by which adaptation is specified and managed. The key research challenges associated with realizing this vision have been to (1) identify levels of abstractions and system views that are relevant for effective DRE system development that span a variety of DRE architectures, but yet can be tailored to specific product lines [CN02, Sha98b], (2) develop facilities for accumulating and visualizing the results of analysis and for visualizing adaptation of system entities, and (3) identify appropriate mechanisms for modeling crucial QoS aspects, such as real-time and availability attributes and constraints/conflicts between these.

Leveraging meta-modeling technology for scaling and product-line tailoring. We have sought to provide a collection of standardized views that can be universally applied to a variety of DRE architectures, but yet can be tailored to particular architectures and product lines. For example, many DRE component models share common abstraction levels, such as (1) *component interfaces*, where a components ports and external attributes are defined, (2) *component implementations*, which are a lower level of abstraction where the 10-20 classes that make up a component are implemented and associated with its interface ports, (3) *system assemblies*, which are a level higher than component interfaces in which components are connected together by their ports, (4) *subsystems*, which are a level higher than system assembly where components are grouped according various goals (such as functionality or protection), and (5) *deployments*, which are a refinement of system assembly in which the characteristics of underlying platforms are modeled.

Our goal has been to develop modeling strategies these general views to be tailored in a systematic way to particular component models and product lines. These strategies enable tailoring to achieve the distinctions between component deployment models (*e.g.*, the CCM and SCA) that have a similar global structure but differ in the specific format of assembly files. Moreover, our strategies have enabled a particular DRE architecture to be further tailored to incorporate domain attributes and development standards for particular product lines (*e.g.*, a component port may include attributes stating various security policies or encryption strengths for a particular SCA product line, while port attributes for an avionics product line include attributes for rate/priority of execution and middleware communication path attributes).

1.2.3 Model and Code-Level Analyses

As the PCES program began, component middleware technology had been effective in relatively small-scale DRE applications, such as statically configured avionics mission computers with 100s of components configured into a deployment [SR03]. One of our goals in the PCES program was to develop different analysis technologies to provide greater *automation* in the configuration of larger systems and to *increase assurance* of completed systems.

Our work involved expanding existing forms of program-level analysis to include analysis at a variety of levels of system abstraction (most importantly, the model level). Models were annotated with various forms of lightweight semantic annotations that capture crucial system properties. In our vision, models and semantic specifications serve two roles: (1) in *property checking* analyses, our lightweight annotations serve as specifications against which a system can be analyzed, and (2) in *property discovery* analyses, they serve as descriptions of properties that are discovered during analyses and which are used to guide system synthesis, configuration and customization.

There has been a flurry of work in recent years on techniques for verifying lightweight functional properties at the program-level [BMMR01, BHPV00, CDH⁺00] but these techniques have yet to be applied in a systematic way to the multiple levels of abstraction in component-based middleware. We have been able to develop approaches for providing higher assurance for DRE systems by adapting a variety of these techniques to DRE abstraction layers. For example, we were able to demonstrate how the global temporal properties (imposing constraints on event orderings etc.) could be verified at various DRE abstraction levels by building on our previous work on verification of Java programs. Specifically, we have shown how the complex semantics of an RT-CORBA event service can be captured using our domain-specific model-checking technology [RDH03] and then used to check temporal properties at the assembly layer for avionics systems [WDMBDJH⁺03]. We have also been able to verify the lightweight annotations describing dependences between components by adapting our Java slicing infrastructure [HCD⁺99] and assembly-level dependence analysis techniques [HDD⁺03].

1.3 Overview of Our DARPA PCES Focus, Results, and Deliverables

The goal of our PCES work has been to show how model-integrated computing and adaptive and flexible middleware frameworks can be applied for defining, analyzing, generating, and customizing large-scale high-assurance, high-performance DRE systems – thus, effectively addressing the challenges described above. Specifically, we aim to provide development frameworks that contain as a centerpiece a variety of forms of software models. Software models provide abstract representations of software structure and behavior that allow developers and analysis tools to focus only on the software features exposed in the model without being overwhelmed by a myriad of low-level details. A primary activity of this project is to take concepts from software industry standards such as the Object Management Group’s “Model-driven Architecture” and adapt these to address challenges particular to military DRE systems.

Project Centerpiece To validate the technologies we developed, we have built a model-integrated development environment called Cadena. Cadena provides a variety of capabilities for model-driven implementation and analysis of component middleware systems. Throughout the course of the PCES project, we have demonstrated that Cadena can dramatically reduce the effort required to construct component-based systems in the context of product-line architectures such as that used for Boeing’s Bold Stroke avionics mission-control software. Moreover, Cadena’s verification and “correct-by-construction” modeling techniques provide increased confidence in the safety and correctness of the resulting system. In Section 4 of this report, we describe the results of using and evaluating Cadena in the following contexts associated with the PCES program.

- The PCES Avionics Open Experimental Platform (OEP) – Working with the PCES OEP providers from Boeing, we defined a broad collection of process metrics (e.g., time required to carry out various stages of system develop, time to locate and repair defects, time to generate implementation code, etc.). In experiments carried out by Boeing engineers, Cadena was able to dramatically reduce the time and effort required to develop different classes of systems (compared to baseline measured efforts provided by Boeing).
- Lockheed Martin / Vanderbilt University (VU) collaboration – Together with researchers from VU, we worked with Lockheed Martin engineers to build and apply an integrated model-driven development tool chain that included Cadena as well as the CoSMIC modeling tool built by VU researchers. This tool chain was used to experiment with model-driven development techniques in the context of using CCM to construct distributed control systems relevant for the Highly Mobile Artillery Rocket System (HIMARS) developed by Lockheed Martin.
- PCES Capstone Demo – Cadena served as the model-driven development platform used to integrate a variety of Real-Time Java technologies that implemented the mission control software for a pair of Unmanned Air

Vehicles (UAVs). The Capstone Demo demonstrated the ability of model-driven development technologies as realized in Cadena to rapidly develop and configure RT-Java based mission critical software.

Deliverables PCES funding has allowed our research group to produce an extraordinary collection of high-quality software engineering tools – each of which is a world-leader in innovations within the associated technology spaces. Section 7 overviews these tools:

- Cadena (described above) – a model-driven development and analysis environment for building component-based systems,
- Indus – a sophisticated analysis and dependency engine for Java,
- Bogor – a extensible software model checking framework capable of supporting domain-specific model checking extensions such as those needed to model check properties of avionics systems deployed on real-time middleware.

During the June 2004 – June 2005 time period, these tools have been downloaded over 3000 times by academic researchers and industrial engineers around the world.

Dissemination of Research Results During the PCES project, we have published over 40 research papers in journals, invited book chapters, conferences, and workshops (a full listing is given in Section 5). During this time, KSU PCES PIs have also given a number of invited talks and tutorials at top-ranking international conferences and workshops (a full listing is given in Section 6).

2 Cadena

There is a wide body of literature dealing with the theory of modeling distributed systems and automated analysis of high-level state-based models using state-space exploration techniques such as model-checking. However, despite the popularity of component-based frameworks and their potential to be utilized in mission- and safety-critical applications, relatively little has been done to scale up these analysis techniques for the purpose of providing automated analysis tools for component frameworks. This is particularly the case with CCM – partly due to the fact that the CCM specification as part of CORBA 3.0 has only recently been finalized. Popular tools such as Rational Rose do not even provide design support for CCM yet.

To investigate the effectiveness of a variety of behavioral analysis techniques for component-based systems, we have built *Cadena*¹ – an integrated development environment for building high-assurance CCM-based systems. Cadena provides facilities for defining component types using CCM IDL, specifying dependency information and transition system semantics for these types, assembling systems from CCM components, visualizing various dependence relationships between components, specifying and verifying correctness properties of models of CCM systems derived from CCM IDL, component assembly information, and Cadena specifications, and producing CORBA stubs and skeletons implemented in Java. We are applying Cadena to avionics applications built using Boeing’s Bold Stroke framework. Thus, important aspects of this work include using dependence information to help generate real-time and distribution aspects, and modeling the real-time CORBA event service used in Bold Stroke.

The primary technical contributions of this Cadena discussed here are

- a framework for light-weight dependency analysis (with varying levels of precision) of component-based specifications, and
- a framework for extracting checkable transition system models from component-based specifications of systems that use middleware services (such as event services) where extracted models incorporate the threading semantics of the relevant middleware services.

¹“Cadena” is a Spanish word meaning “network”. Cadena is also an acronym for Component Architecture Development ENvironment for Avionics systems.

These particular capabilities were developed in response to requests from Boeing engineers working on Boeing's Bold Stroke avionics middleware infrastructure.

In this presentation we further focus on Boeing's Bold Stroke program, which is an example where CORBA middleware has been embraced in a DRE domain for the reasons outlined above [Sha98a, Sha99, DS99]. Bold Stroke is a product-line based program providing object-oriented mission critical avionics software to a variety of military aircraft produced by the Boeing company. Avionics software acts as the center of mission control for an aircraft pilot. It manages the cockpit displays, navigation and tactical sensors as well as weapon deployments. These complex systems have hard and soft real-time deadlines involving large amounts of periodic and aperiodic processing, and support thousands of operating modes. In addition, the software developed for military aircraft is maintained and updated over the course of many years. Although the development process is repeated for each update, each update aims to preserve as much legacy software as possible to reduce cost and risk. Bold Stroke represents a significant technological advance over Boeing's previous mission computing development practices which were largely assembly code based.

We have been interacting extensively with Bold Stroke engineers who have proposed a variety of interesting challenge problems related to component-based design and analysis. Work on Cadena is driven in large part by a desire to provide solutions to challenge problems related to behavioral analysis. Bold Stroke was initiated before the OMG CCM specification process was underway. Thus, the Bold Stroke component design, is slightly different from CCM, and therefore does not apply the CCM Interface Definition Language (IDL) (now part of OMG CORBA IDL 3.0) to auto-generate component code. In current practice, component developers receive a natural language description of functional and real-time requirements along with UML collaboration diagrams built with Rose showing component interactions, and development begins directly with C++ coding. This means that high-level designs are not tool-leveraged in any way (either for code generation or for automated analysis). Bold Stroke engineers have suggested a number interesting ways that high-level designs could be analyzed for event/data dependency and mode state information for the purpose of inferring distribution, scheduling, and real-time aspects, as well as checking for common design flaws and satisfaction of application specific requirements.

Beyond the particular domain of DRE mission/safety-critical systems, we believe that CCM and other component oriented frameworks are excellent vehicles for injecting light-weight formal methods and sophisticated automated analysis techniques across the entire software development process. In the past, it has often been difficult to get developers to write formal specifications – instead they prefer to move quickly to writing code. We believe that this is because there is little tool support for leveraging such high-level descriptions. In contrast, CCM's IDL (which defines the structure of components) and CCM's component assembly descriptions (which describe how components are connected together) are central to the use of CCM since a large percentage of a system's code is generated directly from these. These high-level descriptions can be leveraged in a number of ways: component connections can be visualized (essentially, UML collaboration diagrams can be autogenerated), useful dependency analysis can be performed at this level, light-weight behavior specifications can be incorporated, and code generation can be tailored to produce code that is more amenable to verification and certification. When applying model-checking techniques, one often struggles to find appropriate system abstraction that make state exploration tractable. CCM descriptions naturally form system abstractions, and by varying annotations on the high-level descriptions (e.g., to expose the state of mode variables, etc.) the system model processed by model-checking techniques can be easily abstracted (to hide state) or refined (to expose more state and more interesting behaviors).

Cadena provides the following capabilities for development of CCM systems.

- A collection of light-weight specification forms that can be attached to IDL to specify mode variable domains, intra-component dependencies, and component state-transition semantics. These forms have a natural refinement order so that useful feedback can be obtained with little annotation effort, and increasing the precision of annotation yields more precise analysis. In addition, Cadena specifications allow developers to specify the same information in different ways, achieving a form of *checkable redundancy* that is useful for exposing design flaws.
- Dependency analysis capabilities allow tracing inter/intra-component event and data dependencies, as well as algorithms for synthesizing dependency-based real-time and distribution aspect information.
- A novel model-checking infrastructure dedicated to event-based inter-component communication via real-time middleware enables system design models (derived from CCM IDL, component-assembly descriptions and annotations) to be model-checked for global system properties.

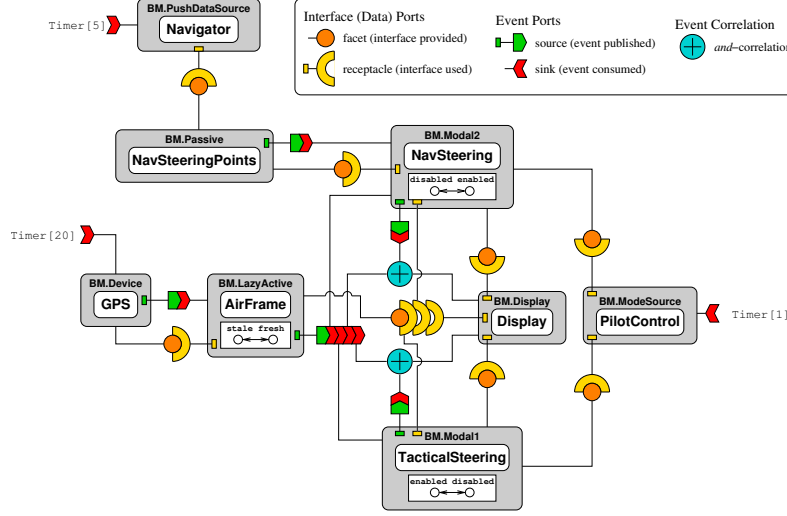


Figure 1: ModalSP – a simple avionics system

- Java component stub and skeleton code generated using the OpenCCM [GOA02] CCM IDL to Java compiler.
- A component assembly framework supporting a variety of visualization and programming tools for developing component connections.
- A CCM deployment facility dedicated to the Boeing Bold Stroke architecture (static component connections with a real-time event-channel) that allows deployment code to be automatically generated.
- The Cadena tools are implemented as plug-ins to IBM’s Eclipse IDE. This provides an end-to-end integrated development environment for CCM-based Java systems.

Several of these facilities are targeted directly to the avionics domain, but Cadena is useful in many respects for CCM development in general. Although Cadena currently emphasizes Java in its back-end facilities, since CCM is language-neutral, Cadena’s front-end design capabilities are not Java dependent. Moreover, back-end capabilities can be easily extended in the future to other languages, for example, C++ using OpenCCM’s [GOA02] planned support for C++. Other development systems such as MetaH [Ves98] support several important aspects for DRE systems that Cadena does not, such as timing and schedulability analysis, reliability and fault analysis, as well as sophisticated deployment strategies. The primary motivation for our work is to build a system that is robust enough for development of real systems with the goal of assessing the effectiveness of applying static analysis, model-checking, and other light-weight formal methods to CCM-based systems.

2.1 CCM Overview and Example

To describe the features of Cadena, we will use as a running example a simple avionics system that shows steering cues on a pilot’s navigational display. The pilot can choose between two different display modes — each mode yields a different set of steering cues. A *tactical* display mode displays cues related to a tactical (i.e., mission) objective. A *navigation* display mode displays cues related to a navigational objective. Cues for the navigation display are derived in part from navigation steering points data that can be entered by the navigator.

Figure 1 presents the CCM architecture for the example system. The system is realized as a collection of components coupled together via interface and event connections. The system include three *modal components* (AirFrame, NavSteering, and TacticalSteering) whose behavior changes depending on the *mode* of the component (the mode of each component will be represented by an attribute with an appropriate enumeration type). Moreover, the system is designed to run on a single processor. For this reason, we refer to the example as *ModalSP*. Input position data is gathered periodically at a rate of 20 Hz in the GPS component and then passed to an intermediate AirFrame component (which in a more realistic system would take position data from a variety of other sensors).

```

#pragma prefix "cadena"
module modalsp {
  interface ReadData {
    readonly attribute any data;
  };

  eventtype TimeOut {};
  eventtype DataAvailable {};

  enum LazyActiveMode {stale, fresh};
  component LazyActive {
    provides ReadData dataOut;
    uses ReadData dataIn;
    publishes DataAvailable outDataAvailable;
    consumes DataAvailable inDataAvailable;
    attribute LazyActiveMode dataStatus;
  };

  enum OnOffMode {enabled, disabled};
  interface ChangeMode {
    attribute OnOffMode modeVar;
  };

  component Modal1 {
    provides ChangeMode modeChange;
    provides ReadData dataOut;
    uses ReadData dataIn;
    publishes DataAvailable outDataAvailable;
    consumes DataAvailable inDataAvailable;
  };
};

```

Figure 2: CCM/Cadena artifacts for ModalSP (excerpts)

Both the NavSteering and TacticalSteering component produce cue data for Display based on air frame position data. The Navigator component polls for inputs from the plane navigator at a rate of 5 Hz and uses those to form NavSteeringPoints data. This data is then used to form navigational steering cues in NavSteering. PilotControl polls for a pilot steering mode at a rate of 1 Hz and enables or disables NavSteering and TacticalSteering accordingly.

Figure 2 gives the CCM IDL that defines the component types LazyActive and Modal1 for the AirFrame and TacticalSteering component instances in Figure 1. CCM components *provide* interfaces to clients on ports referred to as *facets*, and *use* interfaces provided by other clients on ports referred to as *receptacles*. Components *publish* events on ports referred to as *event sources*, and *consume* events on ports referred to as *event sinks*. In the LazyActive component type of Figure 2, dataOut is the name of a facet with interface type ReadData, and dataIn is the name of a receptacle with interface type ReadData. Similarly, inDataAvailable is the name of an event sink of type DataAvailable, and outDataAvailable is the name of an event source of type DataAvailable. Components can also have *attributes* such as modeVar that are used either in component configuration or to represent some other aspect of component state. For an attribute with name *attrname*, the IDL compiler will automatically generate an accessor method *get_attrname* and a mutator method *set_attrname*. If the attribute is declared *readonly* as in the ReadData interface of Figure 2, then only an accessor method is generated².

While CCM allows components to be dynamically created and (dis)connected, Bold Stroke applications follow typical practice in safety/mission-critical systems and employ a static component allocation and configuration policy by creating and connecting components only in a system initialization phase. Dynamic reconfiguration is achieved by including components whose behavior can be deactivated based on the system mode settings.

The CORBA 3.0 specification does not provide a dedicated language for static system configuration. Cadena provides three languages for describing configurations. Graphical, textual, and forms-based descriptions are synchronized through a single internal form. Figure 3 displays a fragment of the textual Cadena Assembly Description (CAD) for the example system. In CAD, a developer declares the component instances that form a system, along with rate and distribution annotations. For receptacle and event sink ports, a *connect* clause declares a connection between a port of the current instance and a port of the component that provides the interface/event. This follows a

²The name of the accessor/mutator methods are dependent on the IDL to language mapping.

```

system ModalSPScenario {
  import cadena.common, cadena.modalsp;

  Rates 1, 5, 20;          // Hz rate groups
  Locations l1, l2, l3;    // abstract deployment locs
  ...
  Instance AirFrame implements LazyActive on #LALoc {
    connect this.inDataAvailable
      to GPS.outDataAvailable runRate #LARate;
    connect this.dataIn to GPS.dataOut;
  }
  Instance TacticalSteering implements Modal1 on l2 {
    connect this.inDataAvailable
      to AirFrame.outDataAvailable runRate 5;
    connect this.dataIn to AirFrame.dataOut;
  }
  ...
}

```

Figure 3: Cadena Assembly Description for ModalSP (excerpts)

convention that connections are declared on the client-side of an interface/event connection. Each event sink port connection uses the `runRate` clause to indicate which rate group thread should run the event handler upon event dispatch. For each instance, a developer names a location upon which the instance is to be allocated. Location names are not bound to physical locations at this point in the process, but will subsequently be mapped to CORBA specific notions such as containers and nodes at deployment time.

Incomplete specifications and incremental construction are supported by allowing rate and location variables such as `#LALoc` and `#LARate`. These act as place holders, and values for these can be inferred using the non-functional aspect synthesis algorithms presented later. Equality constraints between such variables can also be specified, and the synthesis procedures generate output that conforms to these constraints. A type-checking procedure ensures well-typed connections.

Bold Stroke applications follow a *control-push data-pull* architecture in which data is transferred between data producer and data consumer components in a two step process. First, a data producer (e.g., `TacticalSteering`) publishes a `DataAvailable` event indicating that it has updated some data that is ready to be consumed. Then, when a subscribing data consumer (e.g., `Display`) receives the event, it calls a *get data* accessor method in a facet provided by the supplier to retrieve the data. Thus, threads never block waiting for data to become available, and this simplifies the design of real-time aspects. Under this strategy, component connections come in pairs consisting of an event connection for notification that data is ready, and an interface connection for fetching the data.

The `LazyActive` component type of Figure 2 implements a variant of this strategy to handle situations where a component *C* (e.g., `AirFrame`) depends on data that is updated much more frequently than *C*'s clients require *C*'s data. For example, the `AirFrame` component does not fetch data immediately from GPS when notified, but instead simply sets its `dataStatus` attribute to indicate that its data is stale and notifies its clients (e.g., `TacticalSteering`) that its data is available. When a *get data* call for `AirFrame` data comes from one of its clients, it checks the `dataStatus` attribute to see if its data is fresh, and if it is, it returns it immediately to the calling client. If it is not fresh, it calls the GPS *get data* method, updates its own data with the return GPS data, sets its `dataStatus` to fresh, and returns the new `AirFrame` data to the calling client.

Both `NavSteering` and `TacticalSteering` are *modal components* that have two modes (enabled, disabled). These modes are set by `PilotControl` via `ChangeMode` facets provided by the modal components. When a modal component is disabled, any events received are simply discarded by the component. When enabled, the component responds according to the control-push data-pull strategy (e.g., `TacticalSteering` responds to a `DataAvailable` from `AirFrame` by calling `AirFrame`'s *get data* method).

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation, event communication is factored through a real-time CORBA event channel. Use of such infrastructure is central to Bold Stroke computation because it provides not only a mechanism for communicating events, but also a pool-based threading model, time-triggered periodic events, and event correlation. In order to shield application components from the physical aspects of the system, for product-line flexibility, and for run-time efficiency, all components are *passive* – component methods are run by event-channel threads that dispatch events by calling the event handlers (“push methods” in CORBA terminology) associated with event sink

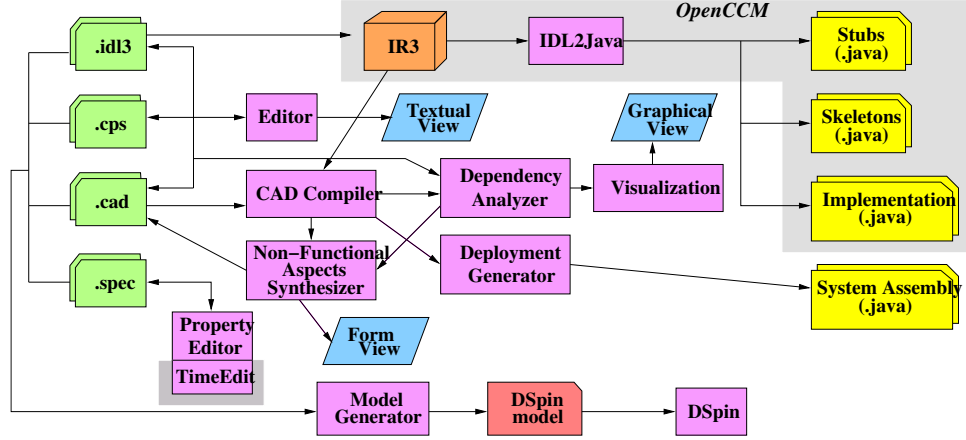


Figure 4: Cadena architecture

ports. The roots of computation are time-triggered events (e.g., events associated with event sinks of Navigator, GPS, and PilotControl) supplied at a specified rates by the event-channel. Dispatching of these events causes the dispatch threads to run the associated handlers which contain methods calls and publishing of subsequent events. In the current Bold Stroke implementation, the event channel thread pool has exactly one thread associated with each rate. As noted earlier in the discussion of Figure 3, each non-time-triggered event port also has a rate specified at configuration time which indicates its *rate group*, i.e., the pool thread that should run the event handler when the event is dispatched.

The event channel also provides event correlation and event filtering mechanisms. In the example system, *and*-correlation is used, for instance, to combine event flows from NavSteering and AirFrame into NavDisplay. The semantics of *and*-correlation on two events e_1 and e_2 is that the event channel waits for an instance of both e_1 and e_2 to be published before creating a notification event that is dispatched to the consumer of the correlation. The semantics of a correlator is defined by an automaton over event traces derived from the correlation expression [Sip02].

Note that CCM IDL captures the interface properties of components – Cadena’s notation for expressing component behavior is presented in the next section.

2.2 Cadena Architecture

Figure 4 displays the internal structure of the Cadena toolset. Cadena projects contain four high-level specification forms: a CORBA 3 IDL file that defines the structure of component types (see Figure 2), a Cadena Property Specification (CPS) file that specifies various aspects of component behavior (see Figure 10), a Cadena Assembly Description (CAD) that specifies the components instances that form the system, the connections between them, along with other real-time and distribution property information (see Figure 3), and a specification file that stores information about the desired correctness properties of the system. These input artifacts are created using customized editors built using Eclipse plug-in facilities. In particular, the CAD format has a textual editor (shown in Figure 5), a graphical editor (shown in Figure 6), and a form-based editor that allows one to easily define different projections of the component assembly (e.g., connections only, distribution and rate assignments only, etc., see Figure 7). The graph structure described by the CAD is the basic data structure that is used by the dependency analyzer (discussed in Section 2.3), the graphical view displayer, and the deployment code generator (which generates Java code to allocate and connect components).

While the assembly is perhaps the most involved part of the system development and hence is supported in Cadena through three different input methodologies, the other aforementioned aspects (namely component definition and behavioral specification) also come with dedicated editing environments. Figure 8 shows the Cadena editor for IDL3 specifications, Figure 9 presents the behavioral specification (CPS) development environment.

Dependence Specifications: Figure 10 displays excerpts of the CPS file for our example system. In a CPS description, developers may declare intracomponent dependencies between ports and simple behavioral descriptions

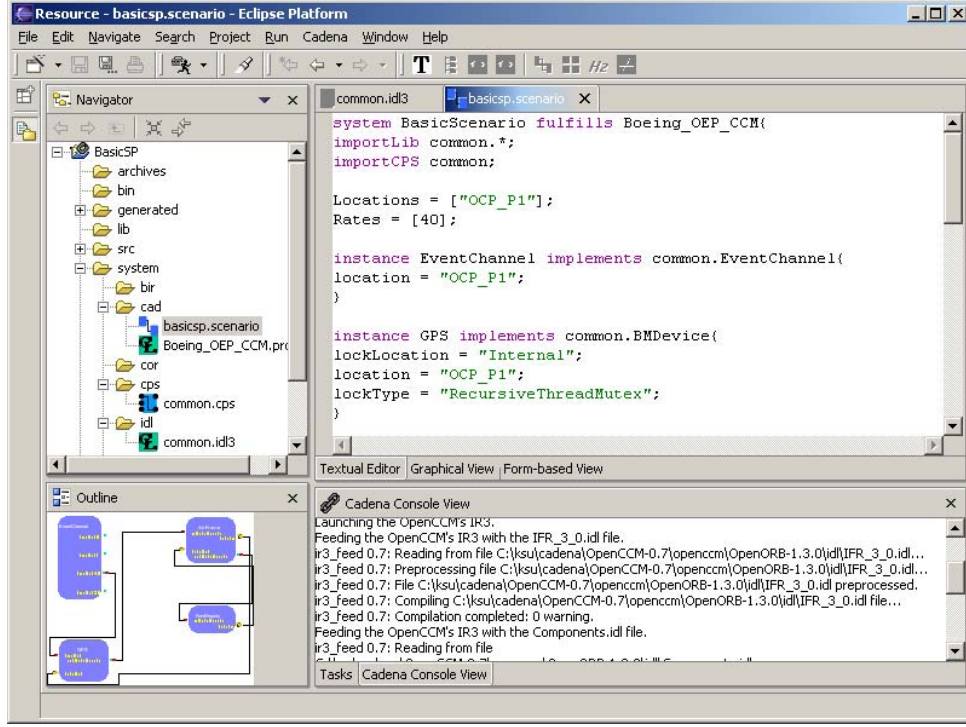


Figure 5: Textual Editor for Scenario Assemblies (CADENA Assembly Description Language)

of a component's event handlers and other methods. The dependence declarations take the form *trigger-port-action* $\rightarrow$ *response-port-action*. For example, Figure 10 declares that consumption of an event on the `inDataAvailable` port of a `LazyActive` component may trigger a publish on the `outDataAvailable` port in both *stale* and *fresh* modes. The absence of a dependence for the `dataOut` port in the *fresh* mode indicates that any call on `dataOut` should not result in an action on any other port.

A `dependencydefault` may have one of two settings: a *none* setting allows developers to start with an empty dependence relation and add new dependencies (i.e., dependencies do not exist except when declared), an *all* setting allows developers to start with a universal dependence relation and then prune dependencies (i.e., by default all possible dependencies between ports exist). In the *all* setting, once a port is mentioned on the left-hand-side, then only declared dependencies apply for that port. For example, for `Modal1` which has the *all* setting, the absence of declarations for the `dataOut` port specifies that the ports (`outDataAvailable` and `dataIn`) *do depend* on `dataOut` (note that is an overapproximation of the actual behavior). Dependencies are pruned in `Modal1` by giving refining declarations such as those for the `modeChange` and `inDataAvailable` ports that list no dependents to the right of the $\rightarrow$.

Behavioral Specifications: Since transition systems for model-checking are generated from behavioral descriptions, their primary purpose is to capture (a) the actions that one wishes to reason about in temporal specifications and (b) simple control-flow relationships between these actions. Cadena supports observable actions such as event publish and consume, method call and return, data flows between system variables, assignments to mode variables. Each behavioral description in the CPS format gives both a data and control abstraction of a component's actual implementation of an event handler or method. Data abstraction is achieved by only exposing concrete values of mode variables (or other application variables with bounded domains). This was motivated by the fact that Bold Stroke engineers are primarily concerned with reasoning about modal behavior at design time since analysis of system modes and mode transitions can be leveraged in several ways. Even though concrete values of other application variables are usually not modeled, data flows between such variables can be captured. For example, `internalData <- dataIn.get_data()`; in the `LazyActive` behavior models a flow from the result of the `get_data()`; method into the `internalData` variable. This may abstract many actual computation steps in the actual implementation. The behavioral language contains simple control structures such as sequencing and

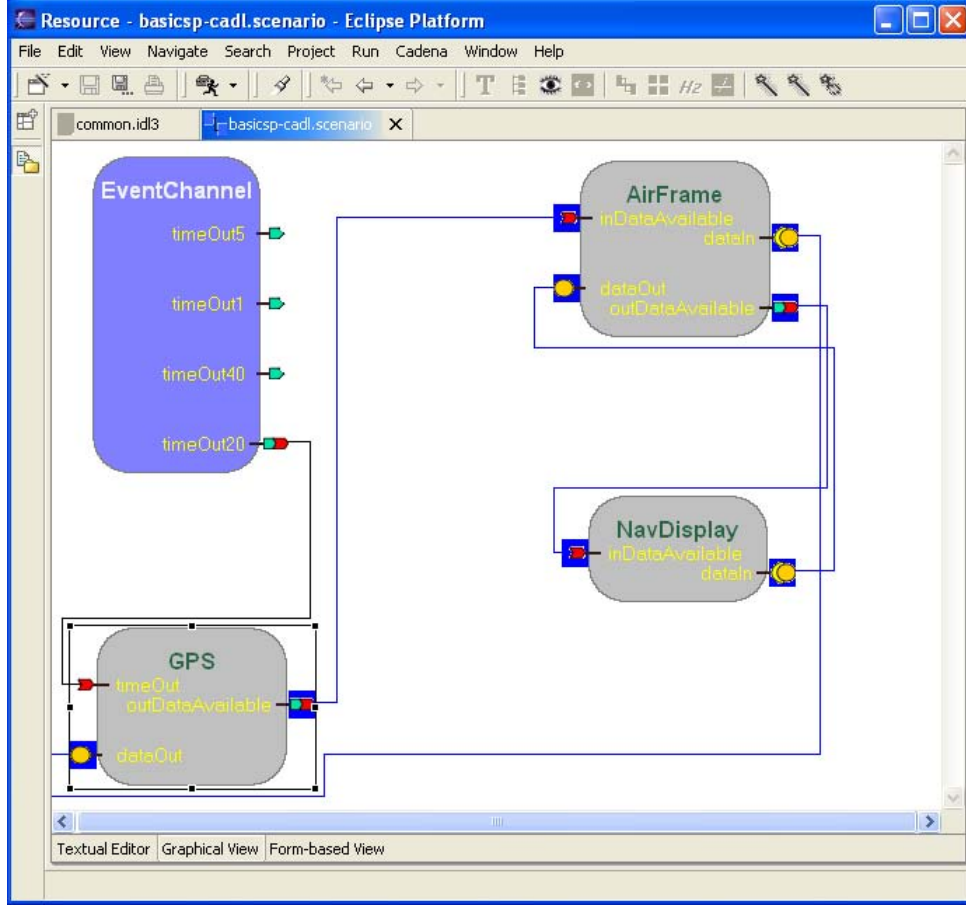


Figure 6: Graphical View/Editor for Scenario Assemblies

conditionals and abstracts control by simply omitting commands from the implementation that one does not wish to observe. Note that dependence information can also be derived from behavioral specifications, and this provides a form of checkable redundancy. The intent is that developers begin with the more light-weight dependence specifications, leverage those, then only incorporate behavioral specifications when model-checking analysis is to be applied.

Code Generation: Cadena uses the OpenCCM tools [GOA02] to generate system implementations. The OMG CORBA 3.0 specification standardizes a strategy for compiling IDL (of which the CCM IDL is part) down to CORBA IDL 2, which can then be translated to an underlying implementation language such as Java or C++. This translation process automatically generates a substantial amount of infrastructure code for tasks such as component creation and connecting and disconnecting ports. The output code contains the usual CORBA *stubs* and *skeletons*, along with skeleton *implementations* of component methods and event handlers. With this code generation, the developer only needs to implement event handlers and methods on provided interfaces. In future work we are exploring the extension of CCM-based code generation strategies to integrate code generation for component handler state-machines and global synchronization policies [DDHM02].

Methodology: When building systems with Cadena, we intend for developers to take the following steps: (1) load a library of domain-specific components and associated CPS specifications, (2) define new project-specific components and associated behavioral CPS specifications, (3) use CAD editors to configure connections between components, (4) use dependency viewer to examine dependencies, (5) use non-functional aspect synthesis tools to attach distribution and rate information (see Section 4.3), (6) specify desired global correctness properties (see Section 5.5), (7) generate a transition system model and model-check correctness properties (see Section 5), and (8) revise system based on feedback from analysis tools.

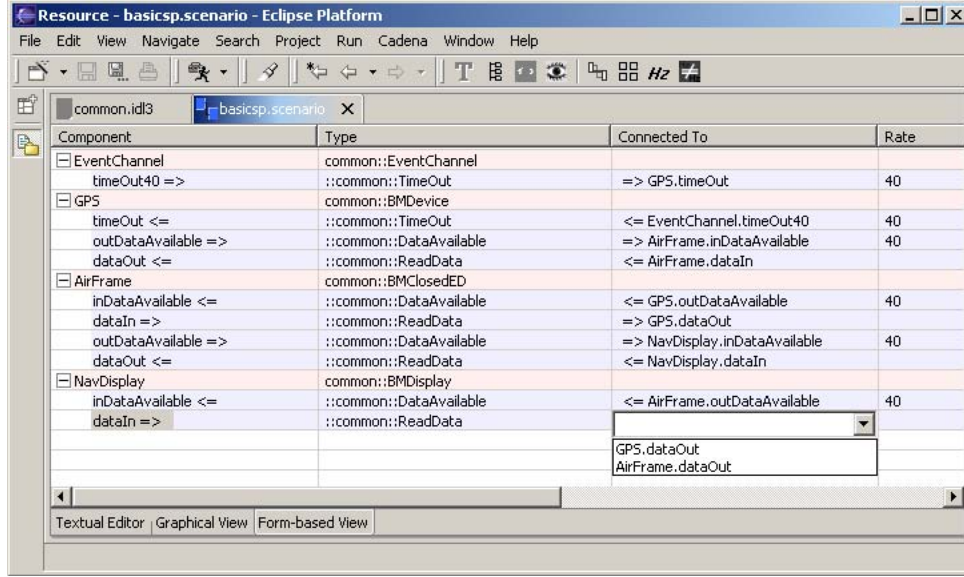


Figure 7: Form-based Editor for Scenario Assemblies

2.3 Dependency Analysis

Even with small systems of around 20-30 components, relationships between components and component dependencies are often hard to determine from visual inspections of textual or graphical component assembly views. Bold Stroke systems can have 1000s of components, and Bold Stroke engineers have identified development of automated support for component dependency analysis and visualization as a high priority. As discussed in the previous section, Cadena provides several different layers of dependency specification and analysis with the goal of enabling incremental construction of dependency specifications – little or no specification effort should still allow useful tool feedback since a fair amount of dependency information (*inter*-component dependences) can be derived from the CAD information. Increasing the details of specifications should yield more precise visualizations and analysis. Here are the steps that we expect developers to take when creating and refining dependence information: (1) give component assembly without CPS dependence information using the global dependence default that all actions on input ports of a component C give rise to actions on all output ports of C , (2) refine by giving dependences without taking into account modal behavior, (3) refine by considering modes in CPS dependence declarations, and (4) refine by giving behavioral descriptions (which capture dependence information via control-flow properties). Bold Stroke developers currently use dependency information manually to determine non-functional aspects such as distribution, connection implementation (synchronous vs. non-synchronous calls), rate group assignment, etc. Cadena leverages partial dependence information to provide automated support for developers.

2.3.1 Basic notions of dependency

Given a component library and component assembly description (along with optional Cadena property specification file), Cadena's port-level dependency module builds a *port dependence graph* $PDG = (N, E)$ where each node $n \in N$ is a component/port pair $i.p$. Edges (i.e., dependences) between PDG nodes arise from two sources: *inter-component dependences* corresponding to port connections specified in component assembly descriptions and *intra-component dependences* captured by CPS declarations in component property specifications. Whether or not intra-component dependences are generated for a particular instance C depends on the *default dependence setting* for C as discussed previously. The default setting is given by the global default dependence declaration unless a component-local default declaration exists.

For inter-component dependences, when there is a connection between $i_1.p_1$ and $i_2.p_2$ in the component assembly description, we say that $i_1.p_1$ is *event dependent* on $i_2.p_2$ (written $i_2.p_2 \xrightarrow{e} i_1.p_1$ – the arrow pointing in the direction of the event flow) if p_2 (resp. p_1) is an event source (resp. sink). Similarly, with the above connection, $i_1.p_1$ is *interface dependent* on $i_2.p_2$ (written $i_2.p_2 \xrightarrow{n} i_1.p_1$) when p_2 (resp. p_1) is a facet (resp. receptacle). For example,

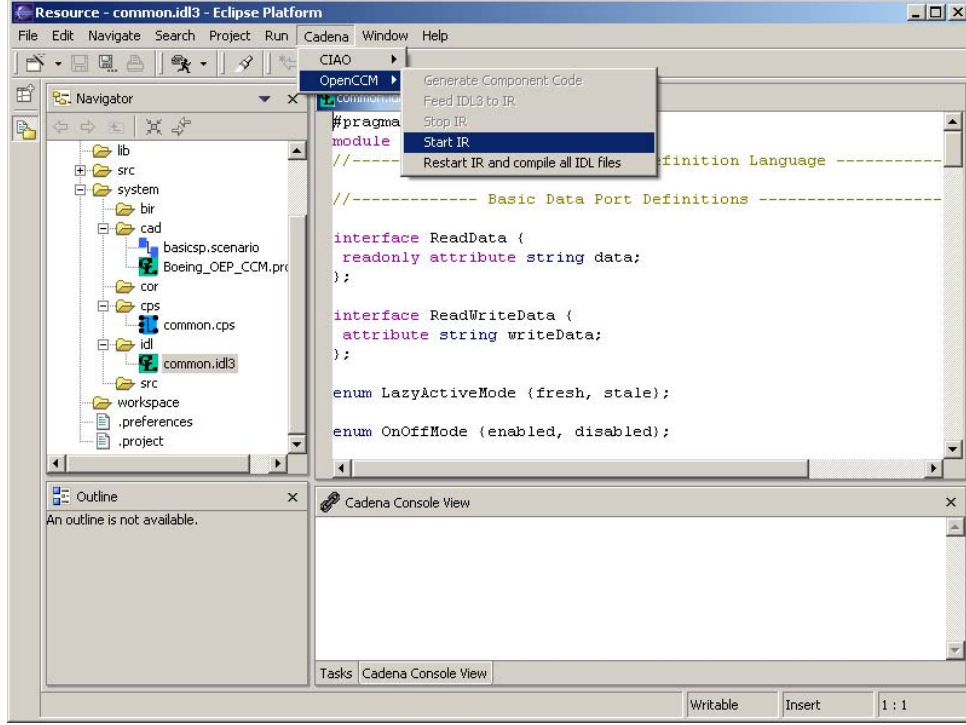


Figure 8: Editor for CORBA Interface Definition Language (IDL) Files

from the CAD information in Figures 1 and 3, we have, e.g., $\text{GPS.outDataAvailable} \xrightarrow{e} \text{AirFrame.inDataAvailable}$ and $\text{GPS.dataOut} \xrightarrow{n} \text{AirFrame.dataIn}$.

For intra-component dependences, for an instance i of component type c , $i.p_1$ is *trigger dependent* on $i.p_2$ (written $i.p_2 \xrightarrow{t} i.p_1$) when either (1) p_2 is declared to trigger p_1 in the CPS for c , or (2) the default dependency status for c is *all* and there exists no trigger declaration for p_2 in the Cadena specification for c .

As with conventional work on dependences, a system slice for a particular point(s) of interest (referred to as the *slicing criterion*) is computed by taking the transitive closure of the PDG from the PDG node(s) corresponding to the slicing criterion. Basic slicing actions provided by Cadena include forward slice, backward slice, and slice intersections.

2.3.2 Mode-aware dependences

To reason about mode-aware dependences, the mode state of the system is captured formally via a mode-state vector m which holds values for one or more mode variables from the system being analyzed. In the ModalSP scenario, it is useful to consider a two-variable mode-state vector that holds the mode state of NavSteering and the mode state of TacticalSteering. Given a PDG $P = (N, E)$ for a system, a modal PDG $P_m = (N_m, E_m)$ for mode-state vector m is formed by setting $N_m = N$ and having E_m include all inter-component edges, but only intra-component edges that are enabled according to m .

Cadena provides mechanisms for collecting a set of mode-state vectors and using these to drive visualization of dependences (i.e., Cadena users can choose to visualize dependences for a particular vector). Mode-state vector sets can be entered explicitly in a form-based view or generated automatically from the state-exploration techniques discussed in the following section. For instance, for the mode-vector mentioned above, it is instructive to have a mode-based dependency view for the three mode-state vectors (disabled, disabled), (enabled, disabled), and (disabled, enabled).

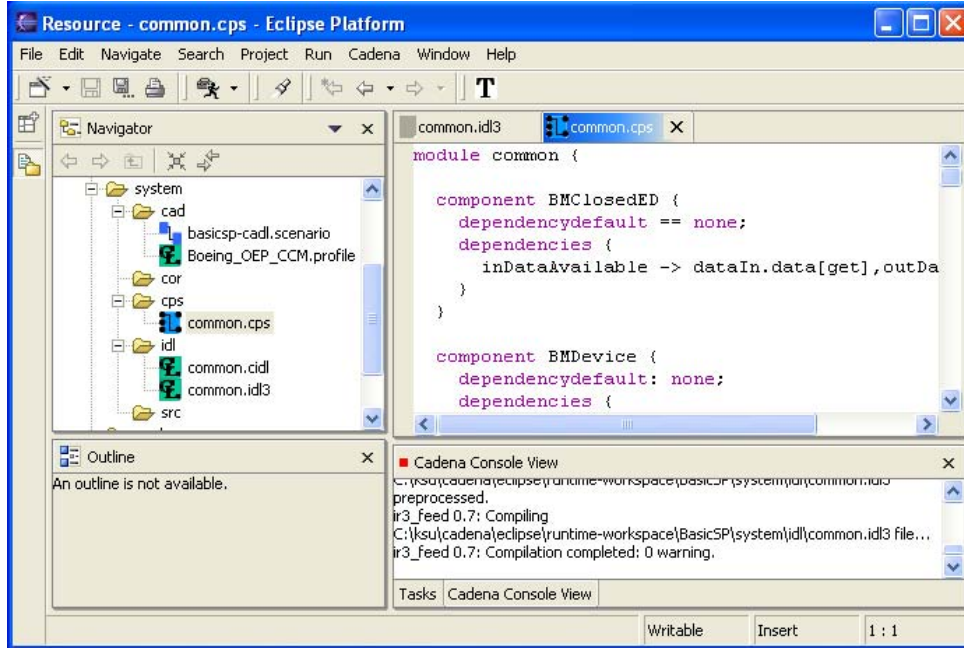


Figure 9: Editor for Component Property Specifications

2.3.3 Cadena support for dependency analysis

Cadena integrates a variety of intra-component aware inter-component dependency-analysis tools, such as *forward slice*, which includes components influenced by a selected component, *backward slice*, i.e. components which influence a selected component, and *chop*, i.e. the intersection of a forward and a backward slice. For finer grained analysis the same operations can be performed on port level and for specific mode-valuations. Finally, the slicing toolset offers circle detection, an analysis useful to avoid feedback loops and oscillation. Figure 11 shows the Cadena display of a forward slice.

2.3.4 Dependency-driven analyses

In the Bold Stroke development process, several non-functional system aspects that are currently designed manually can be aided or even synthesized automatically using the dependence analysis capabilities described above. These include (following the order in which they are carried out) assigning priorities/execution-rates to event consumer ports, appropriately distributing component instances to network nodes, and identifying opportunities for switching asynchronous remote event delivery (the default mechanism) to synchronous local method calls.

Rate Assignment: Automated rate assignment begins by assigning rates to each event consumer port that subscribes to a time-triggered event – the port simply is assigned the rate of the event. Using the results of the dependency analysis above, the process continues by propagating rate information along the PDG in a forwards direction and assigning the propagated rate value to each input and output port encountered. In cases where a port has more than one rate propagated to it (e.g., when event correlation is used, or when two different input ports influence an output port), the highest of the rates is propagated. The process continues until a fixpoint is reached and the resulting rates on event consumer ports bind CAD rate variables.

In the example in Figure 1, automated rate assignment would result in assigning 1Hz to the event consumer of PilotControl, and to the receptacles of PilotControl and the facets of TacticalSteering and NavSteering connected to PilotControl. Similarly, (a) all ports in Navigator, NavSteeringPoints are assigned 5Hz, and the two ports of NavSteering connected to NavSteering are assigned 5Hz, and (b) all ports in GPS and AirFrame are assigned 20Hz. There is now a conflict for the rates on the output ports of NavSteering due to the fact both 5Hz and 20Hz rates are flowing in, so the higher rate of 20Hz is used. The process continues until the remainder of the unassigned ports have a value of 20Hz. Cadena supports rate analysis through specific plug-ins. Figure 12 displays

```

component LazyActive {
  mode dataStatus;
  dependencydefault == none;

  dependencies {
    case dataStatus of {
      stale: inDataAvailable -> outDataAvailable;
      dataOut.get_data(); -> dataIn.get_data();
      fresh: inDataAvailable -> outDataAvailable;
    }
  }

  behavior {
    any internalData;

    dataAvailable.push(_) {
      if (dataStatus == fresh)
        dataStatus = stale;
      outDataAvailable.push(_);
    }

    any dataOut.getData() {
      if (dataStatus == stale) {
        internalData <- dataIn.get_data();
        dataStatus = fresh;
      }
    }
  }
}

return internalData;
}

component Device {
  behavior { ... }
}

component Modal1 {
  mode modeChange.modeVar;
  dependencydefault == all;

  dependencies {
    modeChange ->;
    case modeChange.modeVar of {
      enabled: inDataAvailable
        -> dataIn.get_data(),
        outDataAvailable;
      disabled: inDataAvailable ->;
    }
  }

  behavior { ... }
}

```

Figure 10: Cadena Property Specification (CPS) (excerpts)

an example of dependency analysis cooperating with a rate-seeding heuristic plug-in to discover a mismatch in the rate assignment within a given scenario.

Distribution Determination: The distribution algorithm then uses the rate information gathered above (a) to determine the traffic on the connections between components, and (b) to identify components to be deployed on a common location. In the example, the algorithm would group components closer to their trigger source with the traffic and rate information used as the arbitrating criteria.

In the example, Navigator and PilotControl would be assigned distinct locations, 11 and 12. The rest of the components would be assigned location 13 as the traffic between them is higher assuming all data and event types are of unit size. However, if the traffic on the data connection between Navigator and NavSteeringPoints was higher than the cumulative traffic on other connections on NavSteeringPoints then NavSteeringPoints would be assigned location 11. Although this simplistic example is not realistic, the ability to automatically leverage connection and rate information to provide developers with guidance about component distribution can be a significant advantage for large systems.

Synchronous Dispatch Optimization: For the reduction of asynchronous remote event deliveries, an event delivery between two component instance ports $i_1.p_1$ and $i_2.p_2$ that does not involve correlation can be reduced to a local method call when i_1 and i_2 are co-located and when the rates attached to p_1 and p_2 through the propagation above are the same. In the example, this optimization can be applied to all non-correlated event connections. However, if NavSteeringPoints and NavSteering were assigned different locations in the distribution step above, the optimization would not apply to that connection.

Finally, although we do not implement schedulability analysis in Cadena, we note that Cadena’s dependency specifications (in particular, mode-aware dependence information) can be used to improve static scheduling. Currently, static schedulability analysis in Bold Stroke is based on summing execution costs along call-tree paths deduced from component connections only (i.e., our all dependence mode with no declared dependences). Cadena specifications prune away many infeasible dependences, and therefore prune away infeasible paths that may cause worst-case execution time estimates to be more conservative than necessary. This can sometimes save a surprising amount of development time since systems are often restructured in significant ways to obtain schedulable computations.

2.4 Model Checking Cadena Models with BOGOR

The most powerful analysis methodology of Cadena is the Model Checking integration with our Model Checker BOGOR (described in Section 7.2). In this section we describe our experience and results with applying Model

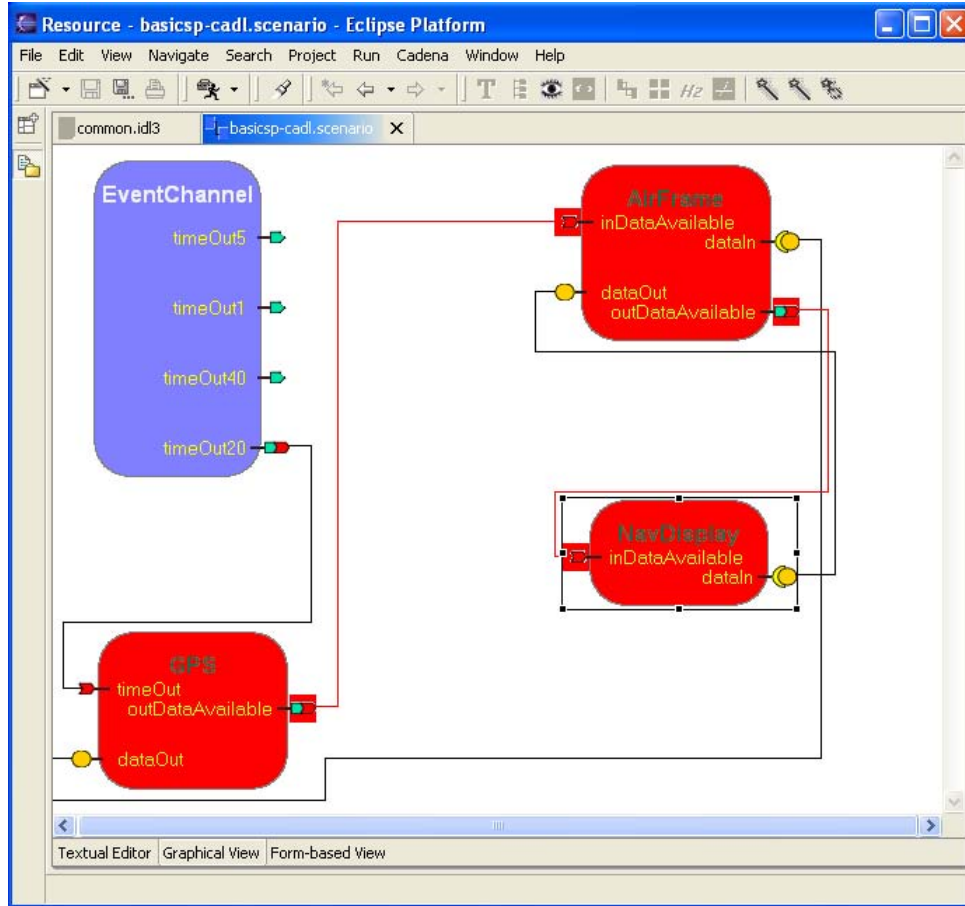


Figure 11: Highlighting Forward Dependencies

Checking to avionic systems.

2.4.1 Modeling approach

Building the approach of Garlan and Khersonsky [GK00] for model checking publish-subscribe systems, we factor Cadena system descriptions into three parts:

- (1) a collection of semantic descriptions for the components that make up the system (these are developer-specified, application dependent and capture aspects (a) and (b) above), and
- (2) a collection of reusable models of run-time event-delivery infrastructure (these are provided to the developer, they are re-used in each application that Cadena supports, and they capture the semantics of inter-component communication identified in aspect (c) above), and
- (3) a collection of connection actions that specifies the connection topology of the components and hooks the component models of part (1) to the middleware models of part (2).

Component models: Component models are defined using a simple transition-based modeling language that is similar to, e.g., Promela [Hol97a] but also includes object references and method calls. Modeling intra-component control-flow as required by aspect (a) above is straightforward using the control constructs of this language. Reasoning about system mode states as required by aspect (b) fits nicely with verification by model-checking since mode variables have small finite domains (e.g., a component is an *enabled* or *disabled* mode). We model such modes using enumerated types in our modeling language. In summary, our component models need only

Component	Type	Connected To	Rate
EventChannel	common::EventChannel		
timeOut40 =>	::common::TimeOut	=> GPS.timeOut	40
GPS	common::BMDevice		
timeOut <=	::common::TimeOut	<= EventChannel.timeOut40	40
outDataAvailable =>	::common::DataAvailable	=> AirFrame.inDataAvailable	5
dataOut <=	::common::ReadData	<= AirFrame.dataIn	
AirFrame	common::BMClosedED		
inDataAvailable <=	::common::DataAvailable	<= GPS.outDataAvailable	5
dataIn =>	::common::ReadData	=> GPS.dataOut	
outDataAvailable =>	::common::DataAvailable	=> NavDisplay.inDataAvailable	40
dataOut <=	::common::ReadData	<= NavDisplay.dataIn	
NavDisplay	common::BMDisplay		
inDataAvailable <=	::common::DataAvailable	<= AirFrame.outDataAvailable	40
dataIn =>	::common::ReadData	=> AirFrame.dataOut	

Figure 12: Detecting Rate Mismatches

include simple control-flow skeletons with transition actions consisting of reads/writes of mode variables, event publish/receives, and method calls to local objects.

Middleware infrastructure models: Modeling inter-component communication is more difficult since the semantics of CORBA communication layers must be captured at a level of abstraction that is fine enough to expose interleavings that can lead to property violations, but also coarse enough to avoid state-space explosion for systems with a large number of components. We define several variants that trade precision for space/time to varying degrees using Bogor’s extension facilities. This involves defining Bogor extensions to represent priority-based event queues and customizing Bogor modules to the particular scheduling strategies used in the RT CORBA middleware. When forming a system model, the developer chooses a particular variant for the middleware model from a library provided by Cadena.

Connection actions: Bogor’s native support for method calls, dynamic object creation, and object references allows components to be connected to the communication layer by passing object references in a manner the closely follows the actual implementation. Accordingly, system initialization is modeled by a sequence of Bogor object-creation statements to create models components and middleware services, followed by a sequence of connection actions that pass appropriate references to establish connectivity.

2.4.2 Real-Time Event Channel

In this section, we give a more detailed description of the CORBA real-time event channel and its elements as shown in Figure 13. This description will be used as a basis of explaining the Bogor event-channel models that Cadena uses when model-checking Bold Stroke systems.

In Bold Stroke applications, even though at a conceptual level component event source ports are connected to event sink ports, in the implementation, event communication is factored through a real-time CORBA event channel. Use of such infrastructure is central to Bold Stroke computation because it provides not only a mechanism for communicating events, but also a pool-based threading model, time-triggered periodic events, and event correlation. In order to shield application components from the physical aspects of the system, for product-line flexibility, and for run-time efficiency, all components are *passive* – component methods are run by event-channel threads that dispatch events by calling the event handlers (“*push methods*” in CORBA terminology) associated with event sink ports. Thus, the event channel layer is the engine of the system in the sense that the threads from its pool drive all the computation of the system.

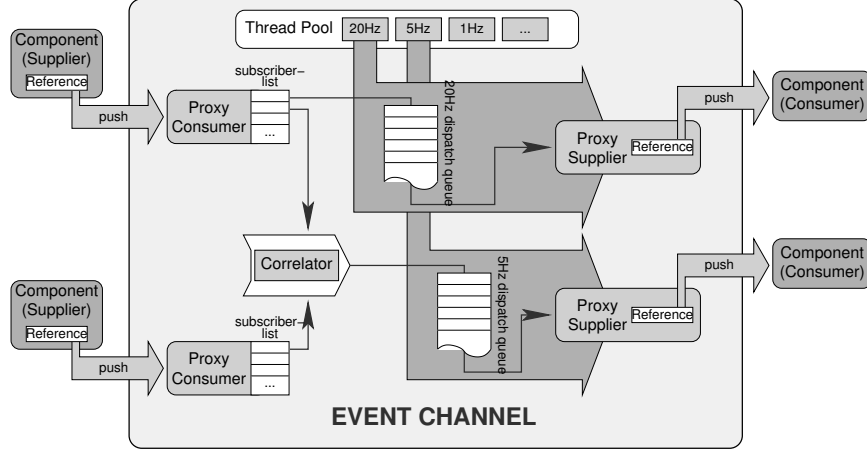


Figure 13: RT CORBA Event Channel

As defined in the CORBA standard an event connection consists of two types of objects, one being the *supplier* (i. e. an event source port), the other the *consumer* (i. e. the event sink port). An object of type consumer must provide a push method, i. e. the event handler method, which takes the event as argument. An object of type supplier stores a reference to a push method. To connect a supplier to a consumer, the supplier's reference is set to point to the consumer's push method. To publish an event e , the supplier simply calls the push method via its reference with the event e as argument (i. e. it “pushes” the event e). Complying to that scheme, the Event Channel offers a *proxy consumer*, i. e. a push method for each supplier to connect to. Similarly, for every consumer the Event Channel provides a *proxy supplier*, a reference to point to the consumer's push method.

This simple reference-to-method pattern allows only one-to-one connections. Since more than one consumer might be interested in the events published by one supplier, the proxy consumer inside the Event Channel features a list of consumers to which an event originating from that supplier will be pushed. This list is called the *subscriber list*, as the consumers *subscribe* to the events of the supplier. This way every consumer and supplier only needs to handle one-to-one connections with the Event Channel, while a multiplexing of the events is done inside the channel.

The Event Channel also provides event correlation and event filtering mechanisms. In the example system of Figure 1, *and*-correlation is used, for instance, to combine event flows from NavSteering and AirFrame into Display. The semantics of *and*-correlation on two events e_1 and e_2 is that the event channel waits for an instance of both e_1 and e_2 to be published before creating a notification event that is dispatched to the consumer of the correlation. The semantics of a correlator is defined by an automaton over event traces derived from the correlation expression [Sip02].

The thread-pool shown in Figure 13 contains the three threads necessary to support the rate groups 20 Hz, 5 Hz, and 1 Hz of the example system of Figure 1. Following rate-monotonic theory, the 20 Hz thread has highest priority, followed by the 5 Hz thread and then the 1 Hz thread. There is an *event dispatch queue* associated with each thread t_r that holds pairs (s, e) where e is the event to be dispatched and s is the reference for an event sink port that is subscribed to the event. Thread t_r dispatches an event e from its queue by running the push method associated with port s via the reference in the related proxy supplier with the event e passed as a parameter.

Periodic computation is initiated by time-triggered events e_r for each rate r (e.g., events associated with event sinks of Navigator, GPS, and PilotControl). At the specified rate r (e.g., at 20 Hz for the GPS event sink), a special internal timer thread (not displayed) places a pair (s, e_r) in r 's queue for each component port s that subscribes to the timeout event e_r . Thread t_r dispatches these events by calling the push methods of subscribers via their proxy suppliers, which in turn may execute methods calls and publish other events to drive further computation.

There are three different paths through the event channel that e can take on its way to a particular subscriber s_k . First, in the normal path, the proxy consumer obtains the reference for s_k and the rate/priority r declared for s_k 's handler for e (recall from the discussion of Figure 3 that each non-time-triggered event sink port also has a rate identifier specified at configuration time that indicates which pool thread should be used to dispatch the event to

which it is subscribed) from its subscriber list and puts the pair (s_k, e) in the queue for t_r .

Second, if s_k is associated with e via an event correlation the pair (s_k, e) is *not* placed in the queue. Rather, the correlator state machine is advanced to account for the publishing of e . If the correlator reaches an accepting state, then a pair (s_k, e_c) is placed in the queue matches the rate declared for s_k 's handler where e_c is a *correlation result event* possibly combines information from one or more events that were correlated.

The third path is an optimization path that short-cuts several steps in the event dispatch process based on the following observation: if there is no correlation associated with (s_k, e) and if subscriber s_k 's handler for e is declared to have the same rate/priority r as the thread t_r that is running e 's publisher, then (s_k, e) will be immediately placed in r 's dispatch queue and the same thread t_r will end up dispatching e . In this case, RT event channel implementation optimizes by having t_r directly call the push method for s_k with e as a parameter – thus, bypassing the queuing/dequeuing of (s_k, e) .

In detail, a trace using example of Figure 1 considering the 20 Hz thread would look as follows. A system interrupt causes the event channel's special timer thread to place a 20 Hz timeout pair (s_k, e_{20}) in the 20 Hz rate group dispatch queue for each 20 Hz time subscriber s_k (in this case, the only subscriber is the timeout port of GPS). Since the 20 Hz queue is no longer empty, the 20 Hz-rate-group thread is started to call the event handler (push method) of GPS. Running the GPS handler for the timeout event reads data from the physical GPS device and issues a `DataAvailable` event, i.e. it calls the according push method in the connected proxy consumer inside the Event Channel. This method then, still executed by the same thread t_{20} , would typically queue the event into the dispatch queues of the subscribing components' thread groups. The subscriber for this `DataAvailable` event from the GPS is the `AirFrame`, which also belongs to the 20 Hz thread group, so in this case the optimization path is used and t_{20} thread directly calls the `inDataAvailable` event handler of `AirFrame`. This handler itself pushes a `DataAvailable` event the consumer proxy associated with the `AirFrame` `outDataAvailable` port. This proxy has a longer list of subscribers: it queues the event into the dispatch queue for the `NavSteering` and for the `TacticalSteering` component, and it forwards the event to the *and*-correlators which also consume events from `NavSteering` and `TacticalSteering` respectively. The state change in the correlators which reflects the incoming event is also executed by the 20 Hz thread, and so is the potential queuing of the correlated event into the Display's rate group's dispatch queue. Since all of these components also run at 20 Hz, the according events are now found in the 20 Hz dispatch queue, and the 20 Hz thread will continue to execute. Assuming that the `TacticalSteering` component is enabled, while the `NavSteering` component is disabled, the push method which handles the incoming event for the `NavSteering` component simply ignores the event, while the `TacticalSteering` calls the `AirFrame`'s facet to fetch the newly available data. Upon this call, the `AirFrame` itself fetches the data from the GPS, switches to fresh-mode and returns the data. After receiving the updated values, `TacticalSteering` issues a `DataAvailable` event itself. Its proxy now forwards this event to the correlator, which already is in a state indicating that the `AirFrame` has already sent his event, so that now a correlated event is queued for the Display. Again in the 20 Hz group this event is the last one in the queue executed by the thread. The thread runs the push method of the Display, which calls the facet of the `TacticalSteering` and receives the new data, and calls the facet of the `AirFrame`, which is in fresh-mode now so that it also immediately returns the new data. The data then is processed and displayed, and the 20 Hz thread ends until the next 20 Hz timeout.

2.4.3 Behavioral Models of Cadena Assemblies

Representing component structure and connections Connections in current Bold Stroke systems are established in an initialization phase, and then remain fixed throughout the lifetime of the system. This means that although connection information must be represented, it does not need to be stored in the state vector. Similarly the interpretation of Cadena models in Bogor can be seen as two phases: first a buildup phase establishes the static part of the system in a single atomic step, then the connected system is checked over the state vector discussed below.

The buildup phase begins with the creation of component instances followed by actions that connect the ports of each instance to ports of other instances (in the case of facets/ receptacles) or the model of the real-time event-channel (in the case of event source/sinks). Figure 14 shows how the Bogor CAD extension supports the buildup of data structures representing components. This extension (not shown) declares two new types `Event` and `Component` (which is used as the type of the BIR `TacticalSteering` variable). Further, the extension defines a number of operations such as `createComponent` and `declareEventSourcePort` that are implemented by Java methods in the extension. Note for example the use of the `bindHandler` operation which declares that

```

CAD.Component TacticalSteering;
enum EnabledDisabled { ENABLED, DISABLED }
EnabledDisabled tacticalSteeringMode;

TacticalSteering := CAD.createComponent("TacticalSteering");
tacticalSteeringMode := EnabledDisabled.ENABLED;
CAD.declareEventSourcePort<EventType>(TacticalSteering,"outDataAvailable",
CAD.declareEventSinkPort<EventType>(TacticalSteering,"inDataAvailable",
    EventType.DataAvailable);
CAD.createField<Data>(TacticalSteering, "ReadData.data");
CAD.bindHandler<EventHandlerEnum>
    (EventHandlerEnum.tacticalSteering_push_inDataAvailable,TacticalSteering,
    "inDataAvailable");
...
CAD.connectEvent(AirFrame, "outDataAvailable",
    TacticalSteering, "inDataAvailable", 20, false);
...

function tacticalSteering_push_inDataAvailable(
    EventHandlerEnum eh, CAD.Event event) {
    Data dat;
    loc loc0: live {}
    when (onOff == EnabledDisabled.ENABLED) do { } goto loc1;
    when !(onOff == EnabledDisabled.ENABLED) do { } return;
    loc loc1: live {} invisible invoke airFrame_facet() goto loc2;
    loc loc2: live{dat}
    when true do {
        dat := CAD.getField<Data>(AirFrame, "ReadData.data");
    } goto loc3;
    loc loc3: live{}
    when true do {
        CAD.setField<Data>(TacticalSteering, "ReadData.data", dat);
    } goto loc4;
    loc loc4: live {}
    invisible invoke pushOffProxy(TacticalSteering, "outDataAvailable", ...);
    return;
}
...

```

Figure 14: Bogor component and assembly descriptions for ModalSP (excerpts)

```

Queue.type<Pair.type<EventHandlerEnum, CAD.Event>> Q5;
...
Q5 := Queue.create<...>(MaxCapacity.QUEUE);
CAD.bindDispatchingQueue<...>(Q5, 5);
...
thread threadgroup5() {
  Pair.type<EventHandlerEnum, CAD.Event> pair;
  EventHandlerEnum handler;
  CAD.Event event;

  loc loc0: live { handler, event } when Queue.size<...>(Q5) > 0
  do invisible {
    pair := Queue.getFront<...>(Q5);
    Queue.dequeue<...>(Q5);
    handler := Pair.first<...>(pair);
    event := Pair.second<...>(pair);
  } goto loc1;
  loc loc1: live {}
  invisible invoke virtual f(handler, event) goto loc0;
}

```

Figure 15: Bogor dispatch queue and thread model for ModalSP (excerpts)

the BIR function displayed at the bottom of Figure 14 is to be used as the event handler for events flowing into the `inDataAvailable` port of `TacticalSteering`.

Below the declaration of the component structure, Figure 14 illustrates the use of the `connectEvent` method. This action causes a Bogor reference to the `inDataAvailable` port of `TacticalSteering` to be added to the subscriber list (recall the discussion of Figure 13 in Section 2.4.2) for the `outDataAvailable` port of `AirFrame`.

When implementing a Bogor extension, one must define a *state manager* that walks over the extension’s state and produces a representation suitable for placing in the model-checker’s state vector. This flexibility can be leveraged in a variety of ways, e.g., to omit various fields from the data vector, to form abstractions of the state, or to build canonical representations necessary for achieving symmetry reductions in the representations of sets [RDH03]. In Cadena models, we use this mechanism to avoid storing the static connection information in the state vector. This also allows us to increase the granularity of actions in initialization and in middleware actions – thus, soundly reducing the number of interleavings. Moreover, traversal of subscriber lists can sometimes be carried out atomically (depending on priorities of threads involved), since there is no chance of interfering updates to subscriber lists once execution begins.

Representing component behavior Event handlers and other methods of CCM components are represented as BIR functions. Figure 14 shows the BIR model of the event handler for the `inDataAvailable` event sink from `Modal1` component type as defined in CPS definition of Figure 10. The transitions capture the handler behavior as defined in the CPS file: if the component is disabled, the handler simply returns, otherwise it fetches data using its `dataIn` port, updates its local data, and then publishes a `dataAvailable` event on its `outDataAvailable` port.

In the example Bold Stroke systems supplied by Boeing, the concrete internal data of components consists exclusively of the values of component mode variables (e.g. the enabled/ disabled values of mode variable `onOff` from component `Modal1`, Figure 10). Boeing engineers abstract away the other data values such as the actual numerical data produced by e.g. GPS devices. Thus, such values are represented by a BIR extension type `Data` (as equivalently by the type `any` in the CPS, Figure 10) that has a single dummy value. Using the state representation mechanism introduced above, component fields of type `Data` are not held in the state vector. This means that component models only contribute the values of mode variables to the state vector.

Representing the real-time event channel The BIR model of the real-time CORBA event service represents the thread-pool, event dispatch queues, and correlators presented in Figure 13 of Section 2.4.2. Recall from Section 2.4.2 that dispatch queues hold event/subscriber pairs (s, e) . In the Bogor model, queues are modeled using `Queue` and `Pair` extensions. Figure 15 illustrates the 5 hertz rate queue of pending event dispatches and the thread, `threadgroup5`, that cyclically dequeues dispatch pairs and invokes the component event handler encoded in each pair (note that pair type declarations are elided (i.e., `<...>`) for improved readability).

Each correlator is represented as a deterministic finite-state automaton whose transition function is encoded as a static transition table. For each correlator, there is a single state variable that holds the current correlator state. Since the structure of correlators is fixed for a given system, the transition tables are not held in the state vector.

Summary of data portion of state-vector To summarize the modeling strategy discussed above, we present the state vector components related to data state of Cadena systems. The *observable state* of a Cadena assembly is comprised of all non-fixed system data. As we have noted above, correlator transition tables, subscriber lists, and component connection information are all fixed and are not considered part of the observable state.

Definition 1 (Cadena Data States) are tuples $(\vec{c}, \vec{r}, \vec{a}, t, p)$ where:

$\vec{c} = (c_1, \dots, c_k)$ stores the data states of component instances, each of which is comprised of a, possibly empty, set of mode attributes as defined by c_i 's component type.

$\vec{r} = (q_{r_1}, \dots, q_{r_n})$ are rate-specific queues of pairs, (c, e) , recording the dispatch of event e to port c .

$\vec{a} = (a_1, \dots, a_l)$ stores the current states of each of the event correlation recognition automata.

t records an abstraction of time used to trigger timeouts.

p records the priority of the current thread being executed.

The initial state is defined to have instance modes set to their initial values, correlation automata set to their start state, rate specific queues to be empty, $t = 0$, and the priority variable is set to the highest priority.

In addition, the values of local variables in component handlers and methods and in the implementation of push methods and rate-specific threads cannot be observed outside their method activation by other threads or by property observables and are also not considered part of the observable state. Local variables are held in the state vector, but only during the corresponding method activations.

Strategies for modeling scheduling and time The behavior of Cadena systems is driven by the triggering of middleware timeouts as described in Section 2.4.2 and is controlled by the scheduling policies of the thread-pool in the real-time event channel. Finding an effective strategy for modeling these timeouts and thread-scheduling is a central issue in the construction of Cadena models.

When analyzing concurrent systems, most model-checkers do not attempt to exploit knowledge of specific timing or scheduling strategies but instead explore all possible interleavings of concurrent actions. If we followed this approach, we would allow timeout events to occur non-deterministically between every system transition and we would allow actions from different threads to be interleaved non-deterministically without consideration of priorities or other scheduling constraints. While such a strategy is *sound* in that it covers all possible system behaviors, the number of states generated makes it impractical for all but the smallest systems.

In the subsections below, we describe several strategies that we use to reduce infeasible interleavings. Each strategy incorporates constraints based on observations about priority scheduling and timeout policies implemented by the real-time middleware.

Priority-based scheduling: Having the model-checker non-deterministically explore interleavings without considering thread priorities obviously introduces schedules that are infeasible in the actual system, e.g., a schedule that continues to execute transitions from a lower priority thread even though a higher-priority thread is enabled.

Inter-rate-group timeout constraints: Having the model-checker non-deterministically generate timeout events introduces schedules that are infeasible in the actual system, e.g., a 5 Hz timeout event should not occur more frequently than a 20 Hz timeout event. We present strategies that reduce infeasible interleavings by taking into account the appropriate relative frequency of timeout events, i.e., by taking into account constraints that exist between timeouts of different rate groups.

Intra-rate-group timeout constraints: Having the model-checker non-deterministically generate timeout events introduces infeasible schedules where a timeout for a rate group r occurs before all events in the current frame for r are dispatched or before the previous timeout from group r is even dispatched. We constrain the generation of time-out events to ensure that timeouts from the same rate group are not triggered “too quickly”.

This strategy constrains the occurrence of timeouts by considering the relative lengths of the real-times frames and constrains scheduling by considering priority information.

Lazy-time with priority scheduling: In addition to the techniques used in the strategy above, this strategy also considers timing estimates for system transition which allows additional infeasible schedules to be removed from consideration.

Representing priority-based scheduling information Bold Stroke systems are priority scheduled based on the results of rate monotonic analysis of a set of harmonic rate groups. The CAD call `connectEvent()`, illustrated in Figure 14, assigns a rate, and hence a priority, to each component handler for a given event. The default non-deterministic scheduling policy in Bogor is implemented by a module that calculates the set of enabled transitions in a given state and passes that set to the state exploration module, which explores each possible outgoing transition. When reporting our experiments, we refer to models that use this strategy as *priority-unaware*. For Cadena models, a Bogor plugin is used that intercepts the set of enabled transitions in a given state, selects the transitions with the highest priority and passes these transitions on to the state exploration module. As expected, this yields dramatic reductions in the state space, as shown in Section 2.4.4, and also improves the precision of the state space since only infeasible schedules are eliminated (i.e., ones on which a lower-priority transition executes when a higher-priority transition is enabled). We refer to models that use this strategy as *priority aware*. Variations of this plugin are used in the following models to allow for interleaving of timeouts with the highest-priority enabled transition.

Representing intra-rate-group timing constraints The treatment of time, t , determines, in part, the fidelity of the model with respect to the real system’s behaviors. If detailed timing information is available one can keep track of time as component actions are executed and use that time value to trigger periodic events. However, even when timing information is not available, one can still reduce the occurrence of timeout events based on both intra- and inter-rate-group constraints.

Intra-rate-group constraints that we consider involve the notion of *frame overrun*. A frame overrun occurs when a timeout event e_r for rate group r occurs before all events e' triggered directly or indirectly by the previous timeout for r are processed by the rate group’s thread t_r . In normal situations, a timeout e_r occurs and is dispatched, other events arrive in the event channel’s dispatch queues (including those associated with r), and thread t_r becomes idle after all events associated with r have been dispatched. The time that t_r remains idle waiting for the next r timeout is called *slack time*. If a system has a frame overrun error, a thread t_r has no slack time – it is unable to finish all of its work before the next timeout e_r arrives.

Note that exploring the state-space of systems where arbitrary frame overruns are modeled results in a huge number of additional system behaviors that would very likely be infeasible if actual timing data were considered (timing data would allow us to conclude that in most cases frame overruns do not occur). While frame overruns are a real source of bugs in Bold Stroke systems, engineers have other tools and methods for detecting these types of errors. Accordingly, we will reduce the state space that we explore using two strategies. The first strategy which we call **no overruns** assumes that no frame overruns occur at all. This is implemented by having the model-checker scheduler only emit a timeout event for rate group r if there are no enabled transitions associated with rate group r – which models the situation where t_r has become idle. The second strategy which we call **limited overruns** is implemented by having the model-checker scheduler only emit a timeout event e_r if there is no other timeout event remaining in the r dispatch queue (but other non-timeout events may still be waiting in the queue for dispatch). Intuitively, that this model includes overruns that only spill over into the very next frame but does not include overruns where processing is ‘late’ by more than one additional frame.

Representing inter-rate-group timing constraints The strategies related to frame overrun in the previous section constrain timeout events by considering when they should occur relative to other timeouts from the *same* rate group. We now describe a strategy which we call the **relative-time (RT)** strategy that constrains the issuing of timeout events by considering when a timeout for r should occur relative to a timeout for a *different* rate group r' . Specifically, we take advantage of the fact that in rate-monotonic scheduling theory (which is used in Bold Stroke systems), the frame associated with a rate can be evenly divided into some whole number of r' -frames for each rate r' that is higher than r . In the example system of Figure 1, the frame of the slowest rate (1 Hz) can be divided into 5 5 Hz frames, and each 5 Hz frame can be divided into 4 20 Hz frames. The longest frame/period (the frame associated with the lowest rate) is called the *hyper-period*.

The **relative-time** model enforces the following constraints related to issuing of timeouts:

- a single timeout is issued for the slowest rate group in the hyper-period,

```

CAD.Component Timer;
...
Timer := CAD.createComponent("Timer");
CAD.declareEventSourcePort<EventType>(Timer,"timeOut5",EventType.TimeOut);
...
thread timerThread() {
  loc loc0: live {}
  when true do { time := (time + 1) % 20; } goto loc0;
}
...
thread timeOutSenderThread() {
  ...
  loc loc1: // 5 Hz timeout case
  when time % (20/5) == 0 do invisible { } goto locInvoke5;
  when time % (20/5) != 0 do invisible { } goto loc2;
  ...
  loc locInvoke5: live {localTime}
  invisible invoke pushOfProxy(Timer, "timeOut5",
                             CAD.createEvent<EventType> (EventType.TimeOut))

  goto loc2;
  ...
  loc loc2: // 1 Hz timeout case
  ...
}

```

Figure 16: Timer and TimeOutSender thread models for ModalSP (excerpts)

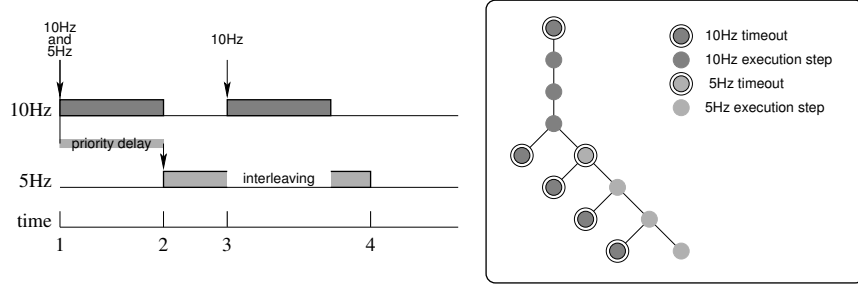


Figure 17: Relative-time Environment

- timeouts for rate groups, r_i and r_j where $r_i > r_j$, are issued such that r_i/r_j timeouts of rate r_i are issued in a r_j frame.

These constraints determine the total number and relative ordering of instances of timeouts that may occur in the hyper-period.

Figure 16 shows the Bogor code for two threads that are used to model this strategy. Thread `timerThread` increments an abstraction of time where each 'tick' (i.e., each increment of the `time` variable) represents the passing of time corresponding to the shortest frame in the system (e.g., in the ModalSP, each tick represents a 20 Hz frame). The `time` variable wraps around every 20 ticks which corresponds to the fact that there are 20 Hz frames in the 1 Hz hyper-period. Thread `timeOutSenderThread` models the behavior of the rate-specific timer threads in the middleware discussed in Section 2.4.2. This thread monitors `time` and when it observes a change in the time value, it passes through a case statement to see which timeout events should be dispatched at that point. Since a time tick represents the period of the shortest frame, a new timeout event for the fastest rate is issued on each pass through the case statement. In our example system, the 5 Hz timeout happens every fourth tick. To represent the occurrence of a timeout, the thread enqueues the timeout event through the standard push call.

From the explanation above, it is clear that the **RT** model only establishes the occurrence of timeouts relative to each other – it does not relate timeout occurrences to the time required by component event handlers and method execution. Thus, it is now important to understand when timeout actions may occur with respect to actions that occur inside of component handlers, i.e., when can these actions be interrupted by timeouts.

To see that the model safely approximates all interleavings of timeouts and component actions (given the constraint on no frame overruns) consider Figure 17. This figure illustrates four points during a system execution which contains 5 Hz and 10 Hz rate processing. The 10 and 5 Hz timeouts are queued together (e.g., at the point 1)

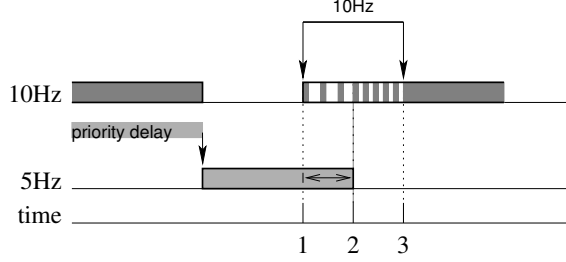


Figure 18: Lazily-timed Environment

since they both have frames that begin at the same point. However, the 10 Hz timeout event is dispatched first due to its higher priority. Once the all the actions associated with 10 Hz component processing complete (e.g., at point 2), the model-checker scheduler begins consideration of lower priority actions and the 5 Hz timeout is dispatched leading to 5 Hz component processing. Our **no overruns** assumption entails that processing the 10 Hz component actions does not require more time than the period of the 10 Hz frame — thus, the next 10 Hz timeout cannot occur before point 2. Since we are not modeling the actual time required for carrying out component actions, it is impossible to determine the relationship between the time required for 5 Hz component action processing (e.g., the duration from point 2 to 4) and the time until the next 10 Hz timeout (e.g., the duration from point 2 to 3). To safely cover all possibilities, we must allow for any relationship between these durations. To model all such relationships, we adapt Bogor’s standard scheduling module to consider all interleavings of enabled timeouts with the enabled transitions. On the right in Figure 17 the interleavings of the 10 Hz timeout and the enabled transitions performed during 5 Hz component processing are illustrated. The first white circle represents the dispatching of the leftmost 10 Hz timeout event. This is followed by black circles representing transitions in 10 Hz component processing: the first branch point represents the choice between the next 10 Hz timeout (on the left) or dispatching the already queued 5 Hz timeout event (on the right). If 5 Hz processing is selected then the choice between the 10 Hz timeout and 5 Hz processing repeats for the next enabled 5 Hz transition illustrated as a grey circle.

Lazily-Timed Components In the **relative-time** model, timeouts are arranged in a proper order and ratio with respect to each other, but there are no constraints that guarantee that, e.g., the interval between time outs is appropriate for the correspond period. This means that the model may have interleavings in which a timeout, e.g., for r_i , occurs prematurely with respect to an action sequence whose duration is less than $period(r_i)$. For example, if the 5 Hz component processing (i.e., from point 2 to 4) in Figure 17 is guaranteed to be less than the time to the next 10 Hz timeout (i.e., point 4 comes before point 3) then the interleavings of 10 Hz timeouts with 5 Hz processing in the **RT** model will be infeasible. The **lazily-timed (LT)** component model addresses this by leveraging worst-case estimates of the running time of components; these will be available for Cadena systems to support rate monotonic analysis. This model can be configured for whatever granularity of timing information is available. Here we consider worst-case timing estimates for event handlers. Conceptually, the estimates are used to determine whether a handler can run without interruption before the next timeout occurs and, if not, the model non-deterministically interleaves action sequences from the handler with timeouts and higher-priority actions that follow from timeouts.

This model modifies the data associated with time to record the intra-hyper-period (IHP) time normalized by the least common factor of all handler durations and timeout periods, the guards in `timeOutSenderThread` from Figure 16 are adjusted accordingly, and each component handler is modified to include an increment of time. Figure 18 illustrates how these increments are performed. It shows the execution of 5 Hz component processing subsequent to completion of 10 Hz processing in a frame. There are two cases: (1) the worst-case time estimate of the 5 Hz processing (i.e., which runs up to point 2) is less than or equal to the next timeout (i.e., timeout occurs at point 3) or (2) it is not (i.e., timeout occurs at point 1 and interrupts the 5 Hz actions). In case (1), the IHP time is incremented by the worst-case timing estimate of the currently running 5 Hz event handler and the state space exploration algorithm proceeds; note that there is no branching in the state space for this case. In case (2), the IHP time is incremented to the next timeout (i.e., point 1), a non-deterministically chosen prefix of the currently running 5 Hz handler is executed, and then the 10 Hz timeout is performed. By choosing a prefix of the handler actions, we are modeling all possible distributions of timing across the actions of the handler. The remaining

portion of the handler is left for the state-space exploration algorithm after the 10 Hz timeout, and subsequent 10 hertz processing is performed. The difference between point 2 and point 1 (i.e., the worst-case execution time of the handler remaining time of the 10 Hz frame) is assigned to that remaining portion as its duration.

This model can be seen as a refinement of the **RT** model. It eliminates interleavings when the timing estimates guarantee that a group of highest-priority enabled transitions are guaranteed to complete before the next timeout. In the example in Figure 17, if the right-most three light-grey circles correspond to a 5 Hz component handler body whose worst-case execution bound is less than the time to the next 10 Hz timeout, then there would be no branching in that portion of the state space (i.e., the lower two left outgoing arcs to 10 Hz timeouts are eliminated).

2.4.4 Experimental Results

Table 1 shows the results of evaluating our strategies using four example systems provided by Boeing engineers. As an example of how to read a system description, the **ModalSP** scenario that we have used as an example has three threads (for rate groups 1 Hz, 5 Hz, and 20 Hz), 8 components, an event correlation (e/c), and 125 events being generated per one second hyper-period (hp).

For each scenario, we give data for five models that incorporate the modeling strategies presented in the previous section.

- **(R)** is the reference model. There is no scheduling policy for the thread groups in the scenario (it is **priority unaware** and has no intra-rate-group timing constraints). Since a completely interleaved execution is infeasible to check, the **relative time** constraints are used though.
- **(RT-1)** uses two policies: **priority aware** scheduling and the **relative time** environment where we implement the **no frame overruns** strategy for the highest-priority thread only.
- **(RT-2)** is like (RT-1), but also assumes there are **no frame overruns** for all threads.
- **(LT)** is like (RT-2) but uses the **lazy time** environment model.

For each example, we collect the number of transitions *trans*, states *states*, time, and memory consumption *mem* at the end of the search. The numbers of transitions and states are both listed because some of steps in the model are marked as invisible (atomic) for which Bogor will not save the states. The experiments were run on a Pentium 4 2.53 GHz with 1.5Gb RAM using the Java 2 Platform. Bogor’s collapse compression [Hol97b] and heap symmetry [Ios02] and process symmetry [BDH02] reductions are used in all of the experiments. Each of the experiments represents a complete exploration of the state-space of the system.

From the table, the state space generally decreases from model (R), (RT-1), (RT-2), to (LT). This shows that by incorporating more knowledge (e.g., the scheduling policy) of the model that is being checked, less states need to be explored. For example, *Medium*, the largest scenario that we have, cannot be model checked using Bogor or our previous dSpin implementation [HDD⁺03] without employing the reduction strategies used in (RT-2) and (LT). For *Basic* the states are the same for model (RT-1) and (RT-2) because it only has a single thread (thus, there is no interleaving). Model (R) has a larger number of states because the lack of constraints allows the timeout to occur even when events associated with the current frame are still being dispatched. Model (LT) has two more states than (RT-2) due to the overhead introduced by the timing transitions.

Bogor runs out of memory checking *ModalSP* (R) (at 3 million states) and *Medium* (RT-1) (at 13 million states). It is interesting that the states for *ModalSP* (R) require more memory than the states for *Medium* (RT-1). This is an effect of the collapse compression that is used. Specifically, there are three threads in *ModalSP* (R), but only two threads in *Medium* (RT-1). In addition, *ModalSP* (R), which has fewer scheduling constraints, allows more interleaving than *Medium* (RT-1). Thus, the collapse compression can save more in *Medium* (RT-1) than *ModalSP* (R), because there are more similar state bit patterns in *Medium* (RT-1) than in *ModalSP* (R).

3 Model-Driven Optimization of Event Service Middleware

Distributed real-time embedded systems often involve a large number of components, distributed across several processing nodes, interacting with one another in complex ways. Given the specification of a system, an important part of the deployment phase is the implementation of the event and data connections between the components

Example System		(R)	(RT-1)	(RT-2)	(LT)
<i>Basic Scenario</i> Threads: 20Hz Components: 3 Events: 2 per .05sec hp	<i>trans</i>	111	42	42	44
	<i>states</i>	20	12	12	14
	<i>time</i>	.16 sec	.11 sec	.09 sec	.11 sec
	<i>mem</i>	.51Mb	.5Mb	.5Mb	.51Mb
<i>Multi-Rate Scenario</i> Threads: 20Hz, 40Hz Components: 6 Events: 6 per .05sec hp	<i>trans</i>	1.36M	7.5K	.98K	.15K
	<i>states</i>	.12M	1.5K	.1K	33
	<i>time</i>	5 min	1.9 sec	.38 sec	.19 sec
	<i>mem</i>	16Mb	.77Mb	.61Mb	.61Mb
<i>ModalSP Scenario</i> Threads: 1Hz, 5Hz, 20Hz Components: 8 (e/c) Events: 125 per 1sec hp	<i>trans</i>	<i>o.m.</i>	.92M	38.2K	6.27K
	<i>states</i>	3M+	20.9K	9.1K	1.56K
	<i>time</i>	<i>o.m.</i>	20 sec	8.59 sec	2.11 sec
	<i>mem</i>	<i>o.m.</i>	4.1Mb	1.61Mb	1.45Mb
<i>Medium Scenario</i> Threads: 1Hz, 20Hz Components: 50 Events: 820 per 1sec hp	<i>trans</i>	<i>o.m.</i>	<i>o.m.</i>	3.79M	.36M
	<i>states</i>	—	13M+	.74M	74.5K
	<i>time</i>	<i>o.m.</i>	<i>o.m.</i>	29 min	3 min
	<i>mem</i>	<i>o.m.</i>	<i>o.m.</i>	71.8 Mb	21.5Mb

Table 1: Experiment Data

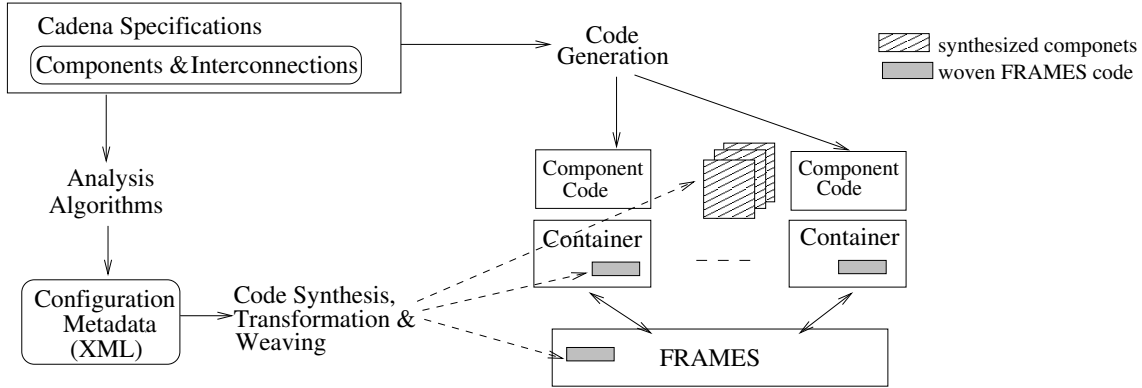


Figure 19: Overview of middleware configuration

via the underlying middleware services. The implementation must not only support the desired communication semantics but also meet stringent performance goals of DRE systems. This may require using different customized versions of a middleware service for different connections. For example, we have been studying the Boeing Bold-Stroke System – a platform to develop avionics applications [Sha99], which uses the Tao’s Real-time event service [HLS97] for event communication. BoldStroke developers have identified several special cases where specialized configuration of the event service is preferred for performance optimization. Incorporating such specialized configurations into the middleware manually can be a tedious task. Therefore, automated techniques are needed that not only generate known specialized configurations but also generate new ones as optimization opportunities are identified. Finally, to take advantage of customization, algorithms are needed that can analyze application’s usage of middleware services to determine when to customize. Such analysis of high-level specifications can provide a systematic way to identify middleware customization opportunities in large-scale DRE systems.

This section describes a FRAMework for Model-driven Event diStribution (FRAMES) for automated customization of event communication middleware. As shown in Figure 19, our model-driven tool chain in FRAMES starts with the application specifications in Cadena. In the configuration and deployment phase, the components, which have been designed in isolation, have to be instantiated and then connected via appropriate middleware services. This phase in Cadena is driven by XML metadata files, which contain information regarding the component instances and their interconnections. From this metadata, Cadena configuration tools generate a system assembly (java) file which contains the code to create the component servers, component homes, and the components. Once

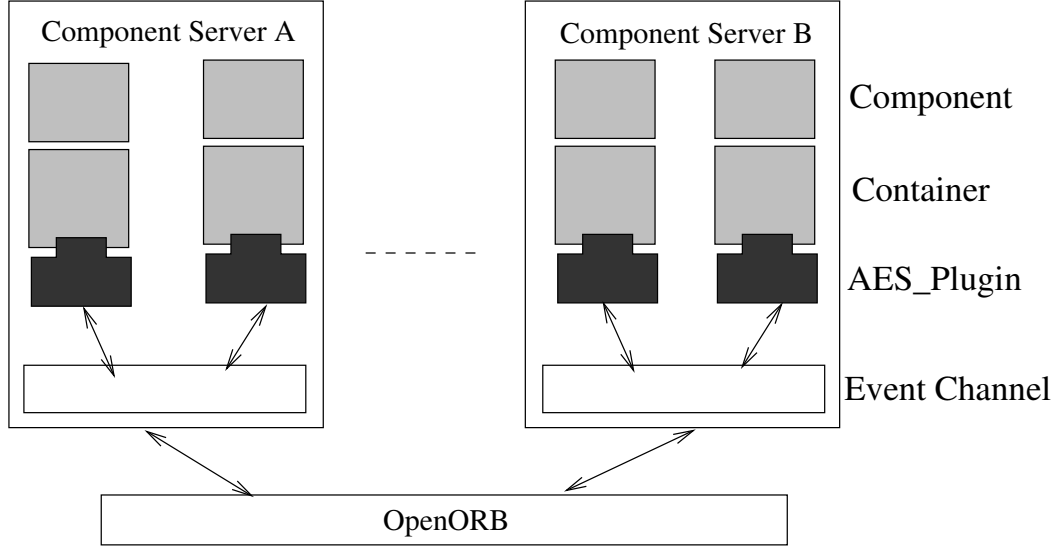


Figure 20: Event Channel Integration

the components are created, we need to implement the event and data connections between them. At this point, analysis algorithms in Cadena analyze the application scenarios to identify when customization may be applicable and generate XML metadata. to generate customized push paths for event notifications using a combination of code synthesis and weaving techniques. The combined use of analysis algorithms and configuration tools enables a large class of possible customizations in the event communication middleware. In the following, we give a brief overview of our approach:

- **Integration of event service into OpenCCM** Communication via an event service is one of the event-communication options available in FRAMES. We have developed Adaptive Event Service (AES), a JAVA based event service, that can configured at deployment time. Furthermore, we have developed a scheme to use AES in OpenCCM containers, whose architecture is shown in Figure 20. There are two main aspects of this architecture. First, in the OpenCCM framework, it is possible to use AES as a stand alone CORBA service. In this case, a single event channel is created and all containers interact with the channel via the ORB. This, however, is inefficient as every notification will be a remote call via the ORB. Therefore, we adopted an architecture wherein an event channel is created in each component server as shown in Figure 20. The components on the same component server communicate via the local event channel (which is more efficient) whereas components on different component servers communicate via gateways. We have developed an algorithm that uses the location information in the CAD file to automatically generate the necessary configuration code to enable this federation of event channels. Second, we want to allow the containers to use any Corba-based event service (such as FACET) at deployment time. To address this issue, we follow a pluggable-module approach wherein each event service provides a plug-in interface module with which the containers can interact (see module AES_Plugin in Figure 20). This approach isolates the containers from the specifics of the middleware services.

- **Configurable Event Service Middleware:** To enable model-driven customization, the middleware itself must be amenable to customization. A push path for event notification from a producer to a consumer may use several event channel features such as subscriber lists, correlation filters and distributed notifications. FRAMES allows these features to be implemented at several places in the CCM code architecture; in the containers or as components (in addition to the middleware level implementations), and these implementations can be used to obtain light-weight push paths. These include the following:

1. **Delivery mechanism:** The possible delivery mechanisms are *Direct Dispatch* and *Event Channel Dispatch*. With Direct Dispatch, the consumer list is maintained in the container of the producer. When an event is pushed by the producer, the container consults this list and makes direct invocation on the consumers. With Event Channel Dispatch, all consumers and producers register with the Event Channel and the consumer lists are maintained in the event channel.

2. *Remote Delivery Mechanism*: The possible options are *Direct Dispatch*, *Component-based Direct Dispatch* and *Gateway-based Event Channel Dispatch*. This option is exercised when there are multiple remote consumers for the same event. With Direct Dispatch (DD), the subscriber lists are maintained in the containers, and each consumer is notified directly via a remote ORB call. With Component-based Direct Dispatch, a new component, called *Gateway_Component*, is deployed in the server in which contains the multiple consumers. When an event is produced, the producer notifies the *Gateway_Component* using a single remote ORB call, and the *Gateway_Component*, in turn, notifies the consumers locally via local calls. The necessary subscriptions to perform this re-routing, and the introduction of the *Gateway_Component* are done automatically by the configuration code. The *Gateway-based Event Channel Dispatch* is similar to *Gateway_Component* approach but at the Event Channel level.

3. *Correlation Implementation*: The possible values are *Event Channel*, *Container*, and *Component-correlation*. With Event Channel implementation, the correlation is done using the correlation support provided by the event channel. With Container-based correlation (CCon), correlation is done by weaving in correlation code in the consumer container. With Component-correlation (CCom), a new component, *Correlation_Component*, is introduced and correlation filter is embedded inside it. This component subscribes to the input events for the correlation expression and produces the events for the consumers. The necessary subscriptions to perform this re-routing, and the introduction of the *Gateway_Component* are done automatically by the configuration code.

FRAMES exposes these different mechanisms as configurable XML metadata options to the application; that is, an application configures the FRAMES via XML metadata specifying the mechanisms to be used for each event connection.

- **Analysis and Customization Tools**: The form-view in Cadena provides each of the options discussed above as selectable options. Although the user can select these options manually, there are several constraints that have to be observed. First, there are some dependencies between the options: For example, if container correlation is selected then the delivery mechanism must be direct dispatching. Some of these constraints are dependent on the underlying CCM implementation. Furthermore, the relative performance of the different options also depends on the underlying implementation. To isolate the designer from such concerns, FRAMES provides an *analysis algorithm* that selects these options automatically. The analysis algorithm uses performance heuristics collected through platform-specific exhaustive testing (discussed below). The algorithms take as input, the system scenario and platform information. In the first pass, the algorithm determines the fan-in and fan-out of the event connections in the system. In the second pass, it analyzes this information and validates it against a set of heuristics to determine the optimal distribution strategy or correlation strategy. This results in selection of the best possible set of options for each event connection. Once the options are selected, Cadena tools generate the necessary metadata and code to introduce any new components, and rewiring of the connections. We have developed configuration tools that performs these actions via a combination of code synthesis, weaving and transformation techniques. For example, to implement correlation in the containers, we synthesize the filter code for the correlation expression, which is then woven into the container code. We use the AspectJ framework for code weaving [KLM⁺97] whereas the code synthesis is pattern-driven. This step does not involve any participation or actions from the designer.

- **Experiments to derive optimization heuristics**: An important part of the customization framework is to determine when to customize. We have conducted a number of experiments evaluating the relative performance of the mechanisms in FRAMES to determine the application contexts/usage in which each mechanism performs better than the others. These have resulted in a set of heuristics which have been inducted into analysis algorithms that analyze an application's event communication structure to identify the best possible mechanism for each event connection. In the following, we give some examples of these experiments. Figure 21 gives the performance of the direct-dispatching (DD) vs gateway-based implementation (in these experiments, OpenCCM creates separate JVMs for each component server, and hence, method calls between component servers are remote calls). As can be seen, for small number of consumers, DD performs better and should be the preferred mechanism. However, as the number of consumers increase, there is a cross-over point at which the gateway-based implementation performs better. This happens because in the gateway scheme, there is only one notification across the server boundary, and the additional cost per consumer is that of a local notification. Figure 22 gives the results of an experiment comparing the performance of the three schemes. The experiment involves varying number of consumers for the same correlated event, and we measure the average time taken to perform correlation, starting at the time at which the first input event is published and ending when all consumers of the correlated event are notified. As can be seen, CCom with DD performs better than AES in all cases. For small number of consumers, the CCon scheme performs better than CCom (since CCom has the overhead of an additional component in the push path). However, in CCon, each consumer performs correlation independently for the same expression whereas in CCom, correlation

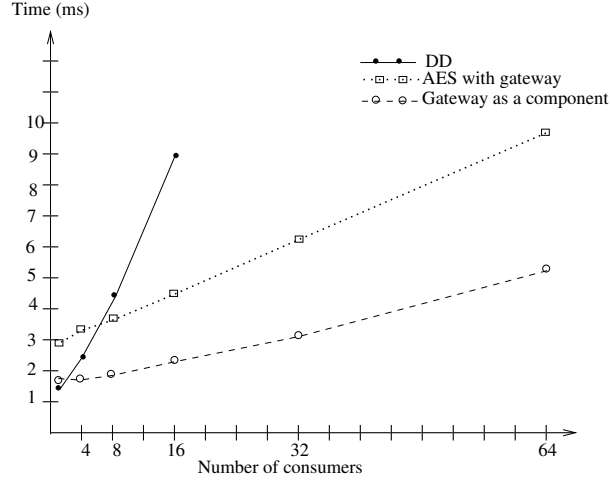


Figure 21: Event Notification with Distribution

is performed once in the correlator component. Hence, as the number of consumers increase, there is a cross-over point at which *CCom* performs better.

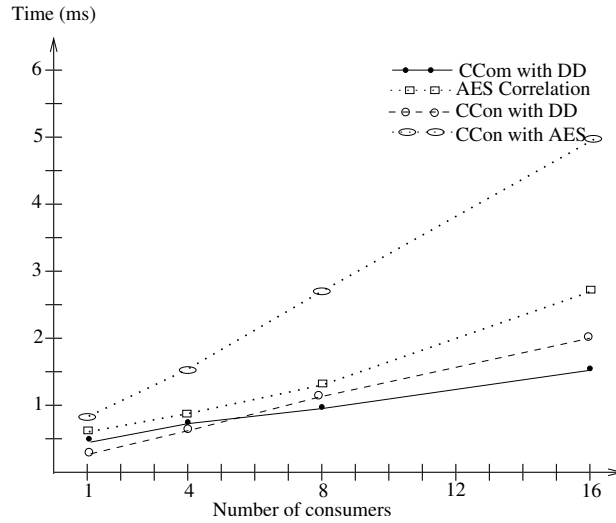


Figure 22: Event Correlation within a component server

4 Summary of Project Results and Impact

4.1 PCES OEP Related Results

The PCES Avionics Open Experimental Platform developed by the Boeing PCES OEP team formed the primary vehicle for our research to demonstrate the effectiveness of technologies that the KSU team developed under PCES. The Avionics OEP presented an idealized development setting and highlighted challenge problems from Boeing's Bold Stroke mission control software program.

The overarching vision that drove our activities on the avionics OEP centered around providing support for capturing and reasoning automatically about dependency and behavioral information present in design artifacts

(such as UML collaboration diagrams) that describe OEP scenarios and XML configuration files.

As noted in the MoBIES/PCES Challenge Problems document, the baseline Boeing current development process,

- did not utilize any automated design-time support for reasoning about event/invoke dependencies and high-level behavior, and
- did not include a formal link between UML artifacts (e.g., the sequence and collaboration diagrams that are currently used for documentation in Bold Stroke development) and system configuration information (e.g., as presented in the XML configuration file) or actual component code.

Avionics OEP documents and personnel emphasize repeatedly that providing effective and scalable support for the items above would represent a significant advance for Bold Stroke development. For example, automated visualization/reasoning for dependencies can be used to detect multiple forms of design errors as well as drive important design decisions (e.g., assignment of components to rate groups).

Section 2 summarizes our efforts to provide a collection of tools that enables designers to reason about component dependencies and high-level system behavior, use partial dependency information to guide other design decisions (e.g., regarding distribution, rate group assignments), and automatically generate configuration information (e.g., in the form of existing XML configuration files) from high-level modeling artifacts (e.g., UML sequence/collaboration diagrams).

We were able to successfully demonstrate success in each of these areas when applied to the scenarios distributed in the OEP. Success was demonstrated by showing significant improvements according to both *process metrics* (which captured time/effort required to develop and debug systems) and *performance metrics* (which capture time and adherence to quality-of-service requirements) in the execution of the system.

For example, for process metrics in the MediumSP OEP example, we were able to make the following improvements to the baseline

- Task of Rate Seeding (assigning priorities to event handlers): 2 hours (baseline) to less than 8 seconds in Cadena
- Task of ERM detection (CORBA co-location optimization detection): 1.5 hours (baseline) to less than 8 seconds in Cadena
- Checking for absence of “event cycle” defect: 3 hours (baseline) to less than 8 seconds in Cadena

Overall, Boeing OEP engineers gave high marks to Cadena throughout the PCES program, which led to Cadena being chosen as the design tool to be used in the RT-Java portion of the PCES Capstone Demo (led by Ed Pla from Boeing).

4.2 Lockheed Martin / Vanderbilt / Kansas State collaboration

PCES teams from Lockheed Martin (Egan), Vanderbilt University, and Kansas State University participated in an effort to demonstrate the effectiveness of model-driven development for development contexts and applications of interest to Lockheed Martin. Collaborative efforts centered around integrating KSU’s Cadena MDD framework with VU’s MDD Cosmic framework and using these to drive development of component-based system built on top of the CIAO CCM C++ middleware implementation from VU.

This integrated tool collection was used to construct small-scale component-based systems representative of the types of systems of interest to Lockheed Martin. Development of these smaller systems served several purposes including debugging of the individual tool component and the information exchange formats used to communicate between the tools, evaluating different features of the individual frameworks, and forming a coherent view of end-to-end evaluation using MDD technology as represented by the integrated tool framework.

Building on the progress made on the smaller scale systems, engineers at Lockheed Martin (Dallas) working on the mission control software for the Highly Mobile Artillery Rocket System (HIMARS) used the integrated tool framework with to experiment with re-engineering HIMARS software to incorporate component technology.

4.3 PCES Capstone Demo Participation

The PCES Capstone Demo aimed to demonstrate that effectiveness of adaptive quality-of-service middleware developed on the PCES project along with PCES model-driven development tools. One component of the project focused on demonstrating the viability of RT-Java technology. The RT-Java portion of the demo was led by Ed Pla from Boeing Phantom Works and included teams from Purdue (who provided the RT-Java VM) and Washington University. The RT-Java team was tasked with building the mission control software for two Scan Eagle UAVs using a RT-Java version of Boeing's PRiSM component middleware.

Due to its unique combination of design and analysis capabilities, Cadena was chosen as the MDD tool for the RT-Java portion of the project. Cadena was used by Ed Bla and other RT-Java team participants to design component interfaces, assemble components instances into systems, debug system scenarios, and generate deployment and configuration scripts for the underlying component middleware framework.

4.4 Kansas State / Rockwell Collins Interaction

Rockwell Collins Advanced Technology Center (RC ATC) was subcontracted to KSU during PCES Phase II to assess model-driven development techniques developed by KSU in the context of applications of importance to Rockwell Collins. Furthermore, RC ATC engineers developed a framework for middleware specialization for the purpose of reducing latencies and footprint of middleware for embedded systems by performing optimizing based on specific information about applications running on the middleware.

Like many other companies developing military relevant systems, Rockwell Collins finds high assurance (HA), distributed, real-time, embedded systems (DRE) are amongst the most challenging of software systems to develop and maintain. In contrast to more mainstream, desktop and business applications, HA-DRE software systems must meet stringent performance, space, certification, and safety constraints. Although RC would like to be able to use commercial off the shelf software (COTS) and open source software in RC systems, it is difficult to adapt more mainstream software to RC needs, both for HA-DRE systems in general, and to meet the demands of specific applications and missions.

As a result, RC is interested in investigating programming environments and infrastructure like Cadena that are targeted directly to the specific requirements of HA-DRE systems.

Rockwell-Collins and Kansas State University have been working together for several years exploring the requirements and possible technology solutions for automated software development tools that would allow us to more effectively develop, maintain, and evolve HA-DRE systems. In particular, we have focused on assessing how OMG-compliant middleware and component technology can be adapted to the HA-DRE domain because middleware is increasingly finding its way into the types of systems we build (both military and commercial), and because middleware may be a key integration technology in future multi-vendor avionics systems. Specifically, Rockwell Collins has a number of large programs that could use this technology. The Joint Tactical Radio System (JTRS) program, for instance, is intended to provide a family of software-programmable, hardware-configurable digital radio systems that can take advantage of rapid changes in commercial technology while meeting the varied needs of the U.S. armed forces, NATO, and civilian applications.

Rockwell-Collins has been prominent in all steps of JTRS development, is a member of the Modular Software Radio Consortium that developed the JTRS Software Communication Architecture (SCA), successfully teamed with Boeing to win the Cluster 1 contract, has developed a large number of waveform applications for JTRS, and is responsible for the evolution and OMG standardization of the SCA architecture under a contract to the JTRS Joint Program Office (JPO). Rockwell Collins is also in competition for a number of future awards. This work is particularly relevant since it requires the customization of the SCA and underlying middleware infrastructure to meet the requirements of a wide variety of computing platforms upon which JTRS software will be deployed. These include the types of low-power, low-overhead platforms needed for future, "smart" weapons systems, plus a number of other types of JTRS systems. In addition to JTRS, Rockwell-Collins is also exploring the use of middleware and component technology in commercial aviation systems.

Given this context, the KSU / RC ATC made significant progress on evaluating the suitability of Cadena for addressing a number of the challenges described above. In particular, we sought to investigate the extent to which general purpose component models such as those used in JTRS could be refined to more specific models that capture the resource constraints for particular platforms.

Other aspects of the work sought to address the problem that most middleware frameworks in existence are

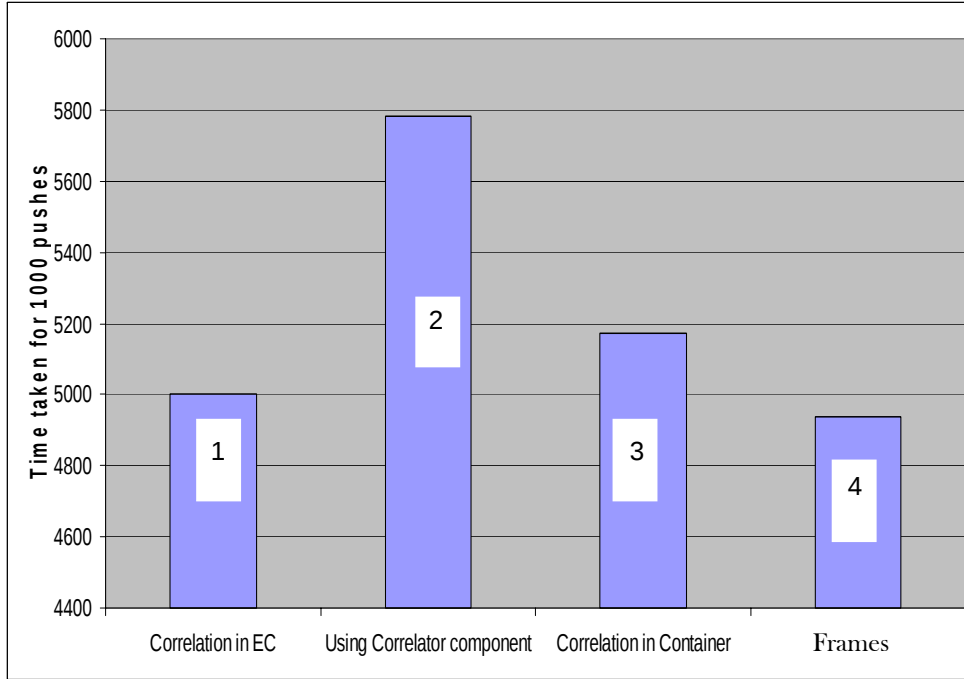


Figure 24: Performance comparison for MediumSP Scenario

consumed by a single consumer and the overhead of the full event channel can be avoided. Hence these events will be tagged as Direct-dispatch. However, in this case, the events to AirFrame are correlated. With direct dispatching as a delivery option, FRAMES algorithms analyze the correlation to determine that the best place to correlate these events would be in the container of AirFrame. Traditionally such events (i.e. involved in correlation) would have been passed through the full Event Channel. However, since AirFrame is the only consumer of that event and Direct-dispatch is traditionally less expensive than an event channel notification, FRAMES opts for container correlation.

Now consider block (2). Track components receive events from the various sensors and compute the current track. However, there is a significant reuse of the correlated events from the same set of sensors. Similarly, Track5, Track8 and Track10 receive events from TrackSensor3 and TrackSensor4. In this case, if container correlation were to be picked, the same correlation would need to be done at the container of several components. Considering all this information, FRAMES determines that in this case the Event Channel might be a better place to perform correlation. Connections in block (3) are similar to block (1), i.e. there is exactly one consumer of the correlated event and hence the best strategy would be to use Direct-Dispatch or ERM with Container correlation.

A sample run of this scenario compared with the other options is shown in Figure 24. Bar-1 indicates the time taken for 1000 pushes when all events are passed through the Event Channel, with Correlation happening only in the Event Channel. Bar-2 shows the time taken when additional components are added to the system to perform correlation. Bar-3 is the time taken when all events are direct dispatched and correlated at the consumer's containers. There is a significant difference because, as per the earlier discussion, there are a significant number of correlations reused in block-2 of MediumSP(Appendix-A), which increases the overhead. Finally, Bar-4 shows the time taken when FRAMES annotates the scenario accordingly. Clearly, for much larger scenarios we can exploit information that adds up to better system performance and predictability.

5 Papers Published with PCES Support

- “Customizing Event Ordering Middleware for Component-based Systems”, Gurdip Singh and Sanghamitra Das, *IEEE International Symposium on Object Oriented Real-time Distributed Computing*, May 2005
- “Constraining Event Flow for Regulation in Pervasive Systems”, P. Shanti Kumar, Q. Zeng and Gurdip Singh, *IEEE International Conference on Pervasive Computing and Communications*, March 2005. pp. 314–318.
- “Model-Driven, Aspect Oriented Synchronization in Component Based Systems”, G. Singh, P. Kumar, and Q. Zeng, *OMG Model Integrated Computing Workshop on Aspect Oriented Modeling*, Feb 2005
- “A New Foundation For Control-Dependence and Slicing for Modern Program Structures”, Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, Matthew B. Dwyer, and John Hatcliff. *Proceedings of the European Symposium On Programming (ESOP’05)*, Edinburgh, Scotland, April 2005, Lecture Notes in Computer Science (3444), pp. 77–93.
- “Kaveri: Delivering Indus Java Program Slicer to Eclipse Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff”. *Proceedings of the International Symposium on Fundamental Approaches to Software Engineering (FASE’05)* Edinburgh, Scotland, April 2005, Lecture Notes in Computer Science (3442), pp. 269–272.
- “Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework”, Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Robby. *Proceedings of 17th Conference on Computer-Aided Verification (CAV 2005)*, Edinburgh, Scotland, July 2005, Lecture Notes in Computer Science (3576), pp. 148–152.
- “Extending JML for Modular Specification and Verification of Multi-Threaded Programs”, Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Robby. *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, Glasgow, Scotland, July 2005, Lecture Notes in Computer Science (3586), pp. 148–152.
- “An Integrated Model-Driven Development Environment for Composing and Validating Distributed Real-Time and Embedded Systems”, Gabriele Trombetti, Aniruddha Gokhale, Douglas C. Schmidt, Jesse Greenwald, John Hatcliff, Georg Jung, Gurdip Singh, In *Model-Driven Software Development*, Beydeda, Sami; Book, Matthias; Gruhn, Volker (Eds.) 2005, ISBN: 3-540-25613-X, pp. 329–362.
- “Analyzing Interaction Orderings with Model Checking”, April 2004. Matthew B. Dwyer, Robby, Oksana Tkachuk, Willem Visser. In the Proceedings of the Nineteenth IEEE International Conference on Automated Software Engineering (ASE 2004).
- “Adaptive Event Communication in Component-based Systems”, Q.Zeng, P. Shanti Kumar and G. Singh, *3rd Workshop on Reflective and Adaptive Middleware*, Oct 2004, pp 201-206
- “Synchronization in CAN-based Embedded Systems”, Y.Su and G. Singh, *Embedded Systems and Application Conference*, June 2004
- “Configurable Event Communication in Cadena”, G. Singh, P. Kumar, and Q. Zeng, *IEEE Real-time and Application Symposium*, May 2004, pp 130-138
- “Exploiting Object Escape and Locking Information in Partial Order Reductions for Concurrent Object-Oriented Programs”, Matthew Dwyer, John Hatcliff, Venkatesh Ranganath, and Robby, *Journal of Formal Methods in System Design*, 25(2), Sep 2004, pp. 199-240.
- “A Priority Inheritance-based Inversion Control Methodology for General Resource Access Problems”, Liubo Chen, Masaaki Mizuno, and Gurdip Singh, *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 202-210
- “A Structured Approach to Develop Concurrent Programs for a Thread-Pool Model,” Liubo Chen and Masaaki Mizuno, *Proceedings of Parallel and Distributed Computing and Systems*, 2004

- “Translating Java for Multiple Model Checkers: the Bandera Back-End”, Radu Iosif, Matthew Dwyer, and John Hatcliff, to appear in *Formal Methods in System Design*, 2004.
- *Proceedings of the Ninth Annual Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, Hubert Garavel and John Hatcliff (editors) Lecture Notes in Computer Science (LNCS) 2619, Springer-Verlag, 2003.
- “Model-driven Design and Implementation of Distributed Real-time Embedded Systems in Cadena”, G. Singh, J. Hatcliff, M. Dwyer, V. Ranganath, X. Deng, P. Kumar, and Q. Zeng, *OMG Workshop on Distributed Object Computing for Real-time and Embedded Systems*, July 2003.
- “Expressing Checkable Properties of Dynamic Systems: The Bandera Specification Language”, James Corbett, Matthew Dwyer, John Hatcliff, Robby. *Journal of Software Tools for Technology Transfer*, 4(1):34-56, 2002. Springer-Verlag
- “Checking Strong Specifications Using an Extensible Software Model-checking Framework”, Robby, Edwin Rodriguez, Matthew Dwyer, and John Hatcliff, *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, April, 2004. Lecture Notes in Computer Science (2988), pp. 404–420.
- “SyncGen : An Aspect-oriented Framework for Automatically Generating Synchronization Implementations from High-level Specifications” John Hatcliff, William Deng, Matthew Dwyer, Masaaki Mizuno, *Proceedings of the International Conference on Tool and Algorithms for Construction and Analysis of Systems*, April, 2004. Lecture Notes in Computer Science (2988), pp. 158–162.
- “An Event Correlation Framework for the CORBA Component Model”, Georg Jung, John Hatcliff, and Venkatesh Ranganath, *Proceedings of the International Conference on Fundamental Aspects of Software Engineering*, April, 2004. Lecture Notes in Computer Science (2984), pp. 144–159.
- “Cadena : An Integrated Development Environment for Analysis, Synthesis, and Verification of Component-based Systems”, Adam Childs, Jesse Greenwald, Venkatesh Ranganath, Xinhua Deng, Matthew Dwyer, John Hatcliff, Georg Jung, Prashant Shanti Kumar, Gurdip Singh, *Proceedings of the International Conference on Fundamental Aspects of Software Engineering*, April, 2004. Lecture Notes in Computer Science (2984), pp. 160–164.
- “Pruning Interference and Ready Dependence the Slicing Concurrent Java Programs”, Venkatesh Ranganath and John Hatcliff, *Proceedings of the International Conference on Compiler Construction*, April, 2004. Lecture Notes in Computer Science (2985), pp. 39–56.
- “A Flexible Framework for the Estimation of Coverage Metrics in Explicit State Software Model Checking”, Edwin Rodríguez, Matthew B. Dwyer, John Hatcliff, Robby. *Proceedings of the 2004 International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, June 2004. Lecture Notes in Computer Science.
- “Verifying Atomicity Specifications for Concurrent Object-Oriented Software using Model-Checking”, John Hatcliff, Robby, and Matthew Dwyer, *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy. January, 2004.
- “Space Reductions for Model Checking Quasi-cyclic Systems”, Matthew Dwyer, Robby, John Hatcliff, and Xinhua Deng, *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, Lecture Notes in Computer Science, Springer-Verlag, Oct., 2003.
- “Bogor: An Extensible and Highly-Modular Model Checking Framework”, Robby, Matthew B. Dwyer, John Hatcliff. *Proceedings of the 2003 ACM Conference on Foundations of Software Engineering*. Helsinki, Finland, September 2003.
- “Space-Reduction Strategies for Model Checking Dynamic Software”, Robby, Matthew Dwyer, John Hatcliff, and Radu Iosif in *Proceedings of the 2nd Workshop on Software Model Checking*, Electronic Notes in Computer Science, 89.3, June, 2003

- “Model-checking Middleware-based Event-driven Real-time Embedded Software”, William Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, Gurdip Singh. *Proceedings of the 2002 International Conference on Formal Methods for Components and Objects (FMCO 2002)*. Leiden, The Netherlands, November 2002 (invited paper).
- “Slicing and Partial Evaluation of CORBA Component Model Designs for Avionics Systems”, John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, Venkatesh Ranganath, and Robby in Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, June, 2003.
- “Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems”, John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, Venkatesh Prasad. *Proceedings of the International Conference on Software Engineering (ICSE 2003)*. IEEE Press. Portland, Oregon, May 2003.
- “Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs”, Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. *Proceedings of International Conference on Software Engineering (ICSE 2002)*, IEEE Press, May 2002.
- “Enhancing Real-Time Event Service for Synchronization in Object Oriented Distributed Systems” ,G. Singh, B. Maddula and Q. Zeng, *IEEE International Symposium on Object-oriented Real-time Distributed Computing*, April 2002, pp 233-240
- “Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software”, John Hatcliff and Matthew Dwyer. *Proceedings of 12th International Conference on Concurrency Theory (CONCUR 2001)*, August 2001, LNCS 2154, Springer-Verlag, pp. 39 – 58 (invited paper).
- “Tool-supported Program Abstraction for Finite-state Verification”, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Willem Visser, Hongjun Zheng. *Proceedings of the International Conference on Software Engineering (ICSE 2001)*, May 2001, IEEE Press.
- “Foundations of the Bandera Abstraction Tools”, John Hatcliff, Matthew B. Dwyer, Corina S. Pasareanu, Robby. pp. 172 – 203. In “The Essence of Computation – Essays dedicated to Neil Jones”. Lecture Notes in Computer Science 2566. Editors: Torben Mogensen, Hal Sudborough, Dave Schmidt

6 Invited Talks describing PCES-funded Work

- Programming Language Design and Implementation (PLDI 2005). Half-day tutorial on “Domain-specific model-checking with Bogor.” (John Hatcliff, Robby). Chicago, USA, June 2005.
- Estonian Summer School in Computer and System Science (ESSCaSS), August 2004 (John Hatcliff) (one of four invited lectures – 6 hours of lectures on Analysis and Verification of Embedded Software).
- European Joint Conferences on Theory and Practice of Software (ETAPS 2004). Half-day tutorial on “Model-checking Software Systems with Bogor.” (John Hatcliff and Robby). Barcelona, Spain, April 2004.
- 2003 Workshop on Specification and Verification of Component-Based Systems, Helsinki, Finland.
- ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2003) (John Hatcliff) (one of two keynote speakers).
- 2002 ACM Foundations of Software Engineering (FSE 2002). “Model-checking Concurrent Java Software with the Bandera Tool Set” (Matt Dwyer).
- International Symposium on Formal Methods for Components, Objects, and their Implementation (FMCO 2002). (John Hatcliff) Leiden, The Netherlands, November, 2002. (one of fifteen invited research talks)
- Schools on Formal Methods (SFM). September, 2002, Bertino, Italy. “Software Model-checking”. (John Hatcliff) (invited three hour lecture – one of eleven invited lecturers for a one-week international Ph.D. school)
- European Joint Conferences on Theory and Practice of Software (ETAPS 2002). Full-day tutorial on “The Bandera Tool Set for Model-checking Concurrent Java Programs”. (John Hatcliff, Matt Dwyer, and Willem Visser (NASA Ames))
- International Conference on Mathematical Foundations of Programming Language Semantics (MFPS’02). March, 2002, New Orleans, LA. “Model-checking Concurrent Java Software Using the Bandera Tool Set” (John Hatcliff) (one of six key-note talks)
- CONCUR 2001: 12th International Conference on Concurrency Theory, Aalborg, Denmark, August 2001. (John Hatcliff) (one of two invited 1.5 hour tutorials)
- JavaCard Verification Project Meeting, August, 2001. INRIA, France. “Model-checking Concurrent Java Software Using the Bandera Tool Set” (John Hatcliff) (invited talk)

7 Software and Other Deliverables

7.1 Cadena

URL: <http://cadena.projects.cis.ksu.edu/>

The use of component models such as Enterprise Java Beans and the CORBA Component Model (CCM) in application development is expanding rapidly. Even in real-time safety/mission-critical domains, component-based development is beginning to take hold as a mechanism for incorporating non-functional aspects such as real-time, quality-of-service, and distribution.

To form an effective basis for the development of such systems, we have built Cadena—an integrated environment for building and modeling CCM systems. Cadena provides the following capabilities:

- A collection of light-weight specification forms that can be attached to CCM’s component Interface Definition Language (IDL) to specify mode variable domains, intra-component dependencies, and component state-transition semantics. These forms have a natural refinement order so that useful feedback can be obtained with little annotation effort, and increasing the precision of annotation yields more precise analysis. In addition, Cadena specifications allow developers to specify the same information in different ways, achieving a form of checkable redundancy that is useful for exposing design flaws.

- Dependency analysis capabilities allow tracing inter/intra-component event and data dependencies, as well as algorithms for synthesizing dependency-based real-time and distribution aspect information.
- A component assembly framework supporting a variety of visualization and programming tools for developing component connections.
- Integration with both C++ and Java CCM implementations including CIAO (a C++ implementation built on top of the ACE/TAO real-time CORBA implementation) and OpenCCM (a Java implementation that runs on top of a number of open source Java ORBs).
- A novel model-checking infrastructure dedicated to event-based inter-component communication via real-time middleware enables system design models (derived from CCM IDL, component-assembly descriptions and annotations) to be model-checked for global system properties.
- Java component stub and skeleton code generated using the CCM IDL compilers of OpenCCM or CIAO.

The Cadena tools are implemented as plug-ins to IBM's Eclipse IDE. This provides an end-to-end integrated development environment for CCM-based Java systems moving from editing of component definitions and connections information to editing and debugging of auto-generated code templates.

Cadena is currently being used by research engineers at several companies including Boeing and Lockheed-Martin to demonstrate the effectiveness of model-driven component-based product-line development for avionics and command-and-control systems. We are actively collaborating with researchers at the University of Vanderbilt (developers of CIAO) to more effectively support model-driven development of distributed real-time embedded systems.

7.2 Bogor

URL: <http://bogor.projects.cis.ksu.edu/>

Modern computing applications increasingly require concurrent/distributed software systems that are extremely reliable. Unfortunately, current software validation techniques, such as inspections and testing, are failing to provide high levels of assurance of correctness for these systems due to system size and complexity as well as the fundamental difficulties of reasoning about state/event sequences in concurrent behavior. Model-checking techniques (now widely used for hardware verification) hold promise for establishing crucial behavioral properties of complex software because they can automatically check to see if an abstract finite-state transition system model of the software conforms to a given state/event sequence property. If the model fails to satisfy the property, the model-checker gives a counterexample — a path through the model's transitions that violates the property. This can be used to locate and correct the corresponding software defect. Essentially, model checking performs an exhaustive simulation that captures the behavior of all possible thread interleavings in the system model being analyzed.

The promise of model checking technology for finding defects due to unanticipated interleavings in highly concurrent systems has led a number of international corporations and government research labs such as Microsoft, IBM, Lucent, NEC, NASA, and Jet Propulsion Laboratories (JPL) to fund their own software model checking projects. We believe that there are a number of trends both in the requirements of computing systems and in the processes by which they are developed that will drive persons and organizations interested in applying model checking technology to rely increasingly on customization/adaptation of existing tool frameworks or construction of new model checking tools tailored to particular domains. Moreover, past experience has shown that software model checking engines need to provide direct support for features of modern programming languages in order to effectively deal with software systems.

To address these trends, we have built Bogor – an extensible software model checking framework with state of the art software model checking algorithms, visualizations, and user interface designed to support both general purpose and domain-specific software model checking. Although there are many model checkers available, Bogor provides a number novel capabilities that make it especially well-suited for checking properties of a variety of modern software artifacts, for building domain-specific model checking engines, and for supporting teaching of model checking concepts.

- Direct support of features found in concurrent object-oriented languages such as dynamic creation of threads and objects, object inheritance, virtual methods, exceptions, garbage collection, etc.
- Bogor's modeling language can be extended with new primitive types, expressions, and commands associated with a particular domain (e.g., multi-agent systems, avionics, security protocols, etc.) and a particular level of abstraction (e.g., design models, source code, byte code, etc.).
- Bogor's open architecture and well-organized module facility allows new algorithms (e.g., for state-space exploration, state storage, etc) and new optimizations (e.g., heuristic search strategies, domain-specific scheduling, etc.) to be easily swapped in to replace Bogor's default model checking algorithms.
- Bogor has a robust feature-rich graphical interface implemented as a plug-in for Eclipse – an open source and extensible universal tool platform from IBM. This user interface provides mechanisms for collecting and naming different Bogor configurations, specification property collections, and a variety of visualization and navigation facilities.
- Bogor is an excellent pedagogical vehicle for teaching foundations and applications of model checking because it allows students to see clean implementations of basic model checking algorithms and to easily enhance and extend these algorithms in course projects.

In short, Bogor aims to be not only a robust and feature-rich software model checking tool that handles the language constructs found in modern large-scale software system designs and implementations, it also aims to be a model checking framework that enables researchers and engineers to create families of domain-specific model checking engines. Bogor is currently being used in a number of research projects outside of Kansas State including University of Massachusetts-Amherst, University of Nebraska-Lincoln, Brigham Young University, Queens University, and EPFL. Bogor has been downloaded over 1600 times since July 2003.

7.3 Indus

URL: <http://indus.projects.cis.ksu.edu/>

Many Java analysis, visualization, and verification tasks require the compile-time computation of a variety of forms of data flow and control flow properties that correctly capture important information about a program's behavior at run-time. Indus is a flexible static analysis framework that provides a variety of forms of flow analyses for Java including a generic data flow analysis framework for inter-procedural analysis of concurrent Java programs, object-flow/aliasing/points-to analyses for constructing precise call-graphs and thread-graphs, escape analysis to detect when heap-allocated objects remain local to a particular method or thread, lock/monitor analyses, and analyses that detect and record various forms of dependence between program elements such as data dependence, control dependence, and dependences between threads.

Using Indus as a foundation, SAnToS researchers have built a program slicer for full Java that provides various forms of visualization, navigation, and querying of program dependence information. This suite of capabilities is called program slicing because it enables developers to selectively view different cuts or slices of a program that consists of program statements that depend on or influence the slice criterion - a set of program points of particular interest. For example, given a failing assertion statement as a slice criterion, a backward slice will find all program statements upon which the given assertion depends (either through transitive data or control dependences). Indus can also produce program chops – all paths of dependence between two different program statements.

Kaveri is a subproject of Indus that delivers the above slicer as a plugin in IBM's open source Eclipse integrated development environment. Using Kaveri, a developer can perform a variety of queries, slices, and chops on program dependence graphs and visualize and navigate the results in a Java editor in Eclipse.

This dependence produced by Indus/Kaveri can be leveraged in a variety of ways:

- Quick fault isolation in debugging. Given a particular Java statement(s) that may be suspected of a fault, a graphical user interface and flexible query language allows users to easily visualize and navigate through program statements that depend on or are influenced by those statements.

- Security analysis of information flow properties. In essence, the dependence information calculated by our tools describes paths along which data can flow in a system. We are currently working toward a security tool built on top of this framework that would enable one to detect, e.g., flows of high-security information into low-security contexts. This type of reasoning is necessary to achieve the partitioning called for in multi-level security architectures.
- Traceability calculations needed for certification. Certification often requires one to trace code to particular requirements, code to dependent code, etc., and these calculations are often very tedious to perform. The dependence analyses that we have developed seem particularly well-suited for supporting that effort. In particular, we are developing ways in which dependence information from Java code is lifted to the modeling level in Cadena – allowing traceability and other certification activities to be organized and directed at the model level.
- Program specialization and program sub-setting: The Indus slicer can be used to obtain specialized versions of a given program by generating a new version of the original program that contains only classes, methods, and program statements that are necessary to carry out the computation at a select set program elements. This program sub-setting effect is useful for reducing the footprint of deployed systems when the complete functionality of the original system is not required.

While several program slicers exist for C, Indus/Kaveri is currently the only available slicer for Java. Thus, it offers Java developers unique capabilities that can enhance system construction in a variety of ways. We are currently extending Indus/Kaveri to RT-Java and to EJB/J2EE applications where dependences could be chased across distributed connections.

Indus has been downloaded over 1500 times since March 2004, and is being used in companies such as NEC, Fujitsu, MITRE, and IBM Japan.

7.4 Specialization Patterns Catalog

As part of our investigation into strategies for adapting general purpose middleware into implementation dedicated to and optimized for specific embedded platforms, our subcontractors at Rockwell Collins Advanced Technology Center have compiled extensive documentation on how to specialize middleware software through the use of semi-automated source code transformation strategies. Specifically, the *PCES Specialization Patterns Catalog* is a 300+ page compilation of patterns related to the specialization of middleware that is intended to validate the basic premise that significant improvements in performance, reductions in footprint, and other simplifications of platform software can be achieved by automatically specializing this software based on knowledge of the application or applications that use it. In essence, knowledge of the application is captured in a model and used to drive a specialization toolset to produce a version of the platform software that is optimized for its intended use. The types of transformation performed by the toolset are formally defined as domain specific 'patterns', allowing the specialization process to be adapted to many domains, and in ways specified directly by the user community.

The types of applications used to illustrate the approach are distributed, real-time and embedded (e.g., a simple Avionics display system). The type of platform software to be specialized is distributed middleware (i.e., CORBA). The patterns are written in a style similar to that for classic OO design patterns, but with sufficient rigor to allow them to be automated, and with the goal of specializing otherwise general purpose software with respect to some set of properties. Specific patterns are intended to optimize particular properties (e.g., we may have different sets of patterns related to performance, footprint, reliability, safety, etc.). Some of the patterns are also general (apply to nearly all software), while others are domain specific (e.g., assume a knowledge of CORBA).

The patterns in this catalog are intended to be automated using a combination of slicing, refactoring, partial evaluation, and aspects. The catalog includes not only patterns that directly apply these technologies, but other forms of refactoring intended to enable their use (e.g., refactoring the code in a particular way may make it easier to apply another pattern that uses partial evaluation). These concepts are integrated with those of aspect-oriented programming (AOP), which are commonly used to introduce new classes, methods, and 'around advice'.

References

- [Ang03] Angelo Corsaro and Douglas C. Schmidt. The Design and Performance of Real-time Java Middleware. *IEEE Transactions on Parallel and Distributed Systems*, 2003.
- [BDH02] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.
- [BGB⁺00] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [BHPV00] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
- [BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.
- [CDH⁺00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
- [CN02] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.
- [DDHM02] Xinghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis and verification of synchronization in concurrent programs. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.
- [DS99] Bryan Doerr and David Sharp. Freeing product line architectures from execution dependencies. In *Proceedings of the Software Technology Conference*, May 1999.
- [GK00] David Garlan and Serge Khersonsky. Model checking implicit-invocation systems. In *Proceedings of the 10th International Workshop on Software Specification and Design*, November 2000.
- [GOA02] GOAL. The OpenCCM platform. <http://corbaweb.lifl.fr/OpenCCM/>, 2002.
- [HC01] George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.
- [HCD⁺99] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, September 1999.
- [HDD⁺03] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
- [HG96] David Harel and Eran Gery. Executable Object Modeling with Statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, 1996.
- [HLS97] T. Harrison, D. Levine, and D. Schmidt. The design and performance of a real-time corba event service. In *Proceedings of OOPLSA*, 1997.
- [Hol97a] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

- [Hol97b] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, April 1997.
- [HV99] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, Reading, MA, 1999.
- [Ios02] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, April 2002.
- [KLM⁺97] G. Kiczales, J. Lampoing, A. Mendhekar, C. Maeda, C. Lopez, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 1241, 1997.
- [LGG⁺01] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr. Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, pages 625–634. IEEE, April 2001.
- [Lin99] Man Lin. Synthesis of Control Software in a Layered Architecture from Hybrid Automata. In *HSCC*, pages 152–164, 1999.
- [Mat99] Matthew Drazhal. *Rose RealTime – A New Standard for RealTime Modeling: White Paper*. Rational (IBM)., Rose Architect Summer Issue 1999 edition, June 1999.
- [Nos02] Russ Noseworthy. IKE 2 – Implementing the Stateful Distributed Object Paradigm . In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Washington, DC, April 2002. IEEE.
- [Obj01a] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.
- [Obj01b] Object Management Group. *Unified Modeling Language (UML) v1.4*, OMG Document formal/2001-09-67 edition, September 2001.
- [Obj02] Object Management Group. *Real-time CORBA Specification*, OMG Document formal/02-08-02 edition, August 2002.
- [Obj03] Object Management Group. *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 edition, May 2003.
- [RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
- [Sha98a] David Sharp. Reducing avionics software cost through component based product line development. In *Proceedings of the Software Technology Conference*, April 1998.
- [Sha98b] David C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Proceedings of the 10th Annual Software Technology Conference*, April 1998.
- [Sha99] David Sharp. Object oriented avionics software flow policies. In *Proceedings of the 18th AIAA/IEEE Digital Avionics Systems Conference*, October 1999.
- [Sip02] Henny Sipma. Event correlation: A formal approach. Technical Report Draft, Stanford University, July 2002.

- [SK97] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.
- [SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294–324, April 1998.
- [SR03] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.
- [Ves98] Steve Vestal. Metah user’s manual. <http://www.htc.honeywell.com/metah>, 1998.
- [WDMBDJH⁺03] William Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, and Gurdip Singh. Model-checking Middleware-based Event-driven Real-time Embedded Software. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *Lecture Notes in Computer Science*, pages 154–181. Springer, March 2003.
- [WSG⁺03] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Craig Rodrigues, Balachandran Natarajan, Joseph P. Loyall, Richard E. Schantz, and Christopher D. Gill. QoS-enabled Middleware. In Qusay Mahmoud, editor, *Middleware for Communications*. Wiley and Sons, New York, 2003.