Distributed Real-Time Managed Systems:

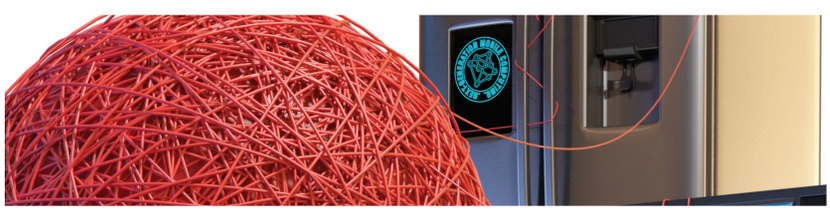
A Model-Driven Distributed Secure Information Architecture Platform for Managed Embedded Systems

Tihamer Levendovszky, Abhishek Dubey, William R. Otte, and Daniel Balasubramanian, Vanderbilt University

Alessandro Coglio, Kestrel Institute

Sandor Nyako, William Emfinger, Pranav Kumar, Aniruddha Gokhale, and Gabor Karsai, Vanderbilt University

// A practical design and runtime solution incorporates modern software development practices and technologies along with novel approaches to address the challenges of effectively managing constrained resources and isolating applications without adverse performance effects. //



MOBILE CLOUD COMPUTING infrastructures supporting the vision of the Internet of Things (IoT)[1] provide services beneficial to our society. For example, a cloud of smartphones can run software that shares the sensing and computing resources of nearby devices, providing increased situational awareness in a disaster zone. A cluster of small, collaborating satellites can provide increased reliability at reduced launch costs for scientific missions; NASA's Edison Demonstration of SmallSat Networks, as well as TanDEM-X, PROBA-3, and Prisma from the European Space Agency, use clusters of small satellites.

Unlike traditional computing clouds, which draw a clear distinction between a cloud provider and user, these roles will be interchangeable in the participating resources in mobile clouds.[2] Additionally, the need to scale up on demand is often the motivation for using a traditional cloud, whereas a mobile embedded cloud is motivated by the need for on demand collaboration.

Table 1 presents associated requirements and challenges that existing cloud computing platforms don't fully address. In this article, we describe an architecture called Distributed Real-time Managed Systems (DREMS; www.isis.vanderbilt.edu/DREMS),[3] which addresses these requirements. It consists of two main parts:

- a design-time tool suite for modeling, analysis, synthesis, integration, debugging, testing, and maintenance of application software built from reusable components and
- a runtime software platform for deploying, managing, and operating application software

0740-7459/14/$31.00 © 2014 IEEE

TABLE 1

A summary of architectural decisions in DREMS.

Requirement	Design principle	Approach
1. Rapid application development, software reuse	Component-based software engineering	Novel component model
2. Multiple application interaction semantics	Separation of concerns	Rich set of component interaction ports with operations scheduled independently
3. Managed concurrency and synchronization for robustness	Single-threaded components	Concurrency managed by OS and middleware, not component business logic
4. Resource management and application isolation with performance guarantees	Spatial and temporal separation	Applications are run in isolated partitions
5. Secure information flows	Multilevel security with multidomain labels, temporal/spatial isolation, and mandatory access control	Architectural support for separation, multilevel security (based on label checking), and constrained information flows
6. Managed and secure application deployment and configuration	Modeling and automation	Model-driven middleware services to provide secure deployment and configuration
7. Producible verified systems	Defects being caught early in the development cycle	Model-based system design and generative development

on a network of embedded, mobile nodes.

The platform reduces the complexity and increases the robustness of software applications by providing reusable technological building blocks in the form of an OS, middleware, and application management services (see Figure 1). For further reading, see the F6 Project Page website: (www.isis.vanderbilt.edu/DREMS and www.kestrel.edu/home/projects/f6) and the Generic Modeling Environment project page (www.isis.vanderbilt.edu/Projects/gme).

Runtime Software Platform: OS and Middleware

DREMS provides a runtime platform for applications in the form of an OS and middleware. The DREMS OS—a set of extensions to the Linux kernel—provides all the critical low-level services to support resource management (including spatial and temporal partitioning of the memory and the CPU), actor management, secure information flows (labeled and managed), and fault tolerance.

Software applications running on DREMS are distributed. To facilitate isolation (requirement 4), the components that make up an application are encapsulated in process-like containers called actors that run concurrently (on the same node) or in parallel (on different nodes). This is similar to the notion of concurrent communicating objects.[4]

Actors are specialized OS processes; they have a persistent identity that allows their transparent migration between computing nodes. They also have strict limits on the resources that they can use. There are two main types of actors: application actors and platform actors. Application actors are built for specific applications, whereas platform actors provide system-level services. The OS guarantees performance isolation between actors of different applications (requirement 4). This is accomplished by

- providing separate, protected address spaces per actor;
- enforcing that a peripheral device can be accessed by only one actor at a time; and
- facilitating temporal isolation between actors by the scheduler.

The temporal isolation is provided via ARINC-653-style partitions[5]—periodically repeating fixed intervals of the CPU's time exclusively assigned to a group of cooperating actors of the same application. The scheduler guarantees that actors in distinct temporal partitions can't inadvertently interfere with each other via CPU usage. (Further details on spatial and temporal isolation, both of which are standard mechanisms, are available elsewhere.[3])

Component Model: Building Blocks for Application Development

To address requirement 1, DREMS uses a component-oriented approach for application development.[6] It's commonly accepted that

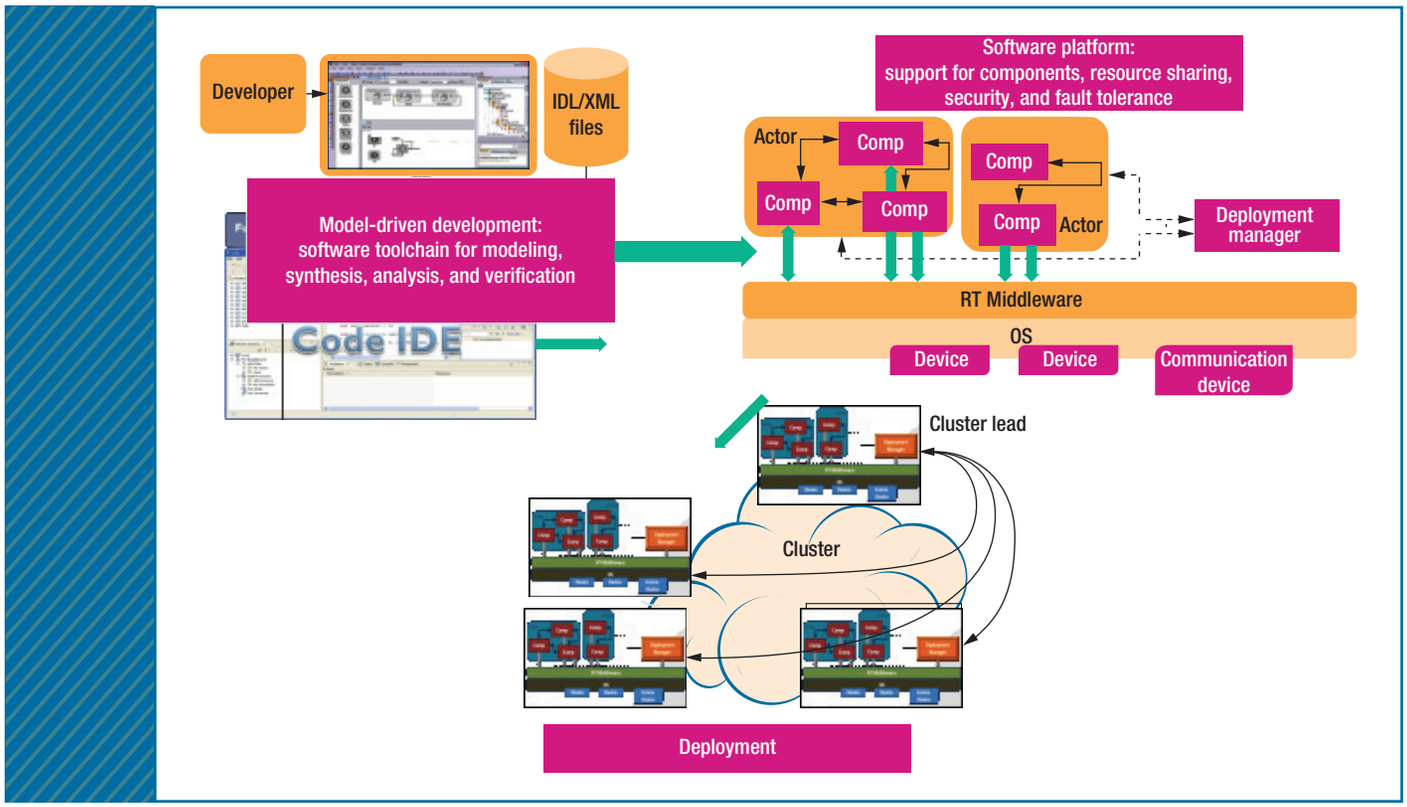


FIGURE 1. DREMS architecture. The top-right portion shows the internals of one node.

component-based software development promotes rapid application development and reuse.[7] Components have identity, have state, support various operations, and interact via ports. A DREMS component supports four basic types of communication ports, providing a range of interaction semantics (requirement 2). First, there are *facets*, which are collections of operations (interfaces) provided by a component, and *receptacles*, which are collections of operations required. These two ports can be used to implement synchronous and asynchronous point-to-point interactions. In addition, *publisher* and *subscriber* ports provide a way for components to interact in a global data space defined over topics. Conceptually, this is similar to the OMG Common Object Request Broker Architecture (CORBA) Component Model.[8]

However, there are some key differences. The DREMS component model provides ports for accessing I/O devices and timers. Ports are implemented using connectors that enable the use of a variety of communication mechanisms,[9] including CORBA and Data Distribution Services (DDS). Furthermore, security using labeled communication is a fundamental part of all component interactions. Another key distinction is the threading model: DREMS meets requirement 3 by enforcing that component activities are scheduled by the middleware as nonpreemptible, single-threaded operations that necessitate no synchronization code from the developer. Note that components do run concurrently.

Secure Transport: A Secure Actor-to-Actor Communication Channel

DREMS provides a security architecture (requirements 4 and 6) based on

- spatial and temporal separation among the actors,
- fine-grained actor privileges that control what system services can be used by an actor,
- assurance that only one actor actively controls a device at a time, and
- a novel communication mechanism among nodes called secure transport, which supports the

exchange of messages among actors according to a multilevel security (MLS) policy.

The combination of separation and MLS guarantee, for example, that an erroneous or malicious actor cannot read information at a higher classification level than its own.

To enforce these rules system wide, application actors aren't permitted to either create new actors or configure secure transport; these activities are performed by the trusted platform actors.

Endpoints and Flows

Actors interact only in controlled ways, which is especially important when they belong to different organizations (such as countries). To exchange messages, actors don't reference each other directly; they reference local endpoints through which messages are sent and received. An endpoint is analogous to a socket handle in traditional networking systems. Endpoints in different actors are connected by flows—that is, "pipes"—through which messages are transferred (see Figure 2). A flow is a connectionless logical association between endpoints: unicast flows connect a source endpoint to a destination endpoint; multicast flows connect a source endpoint to multiple destination endpoints. Both endpoints and flows are created and assigned only by trusted platform actors. Performing message exchanges via endpoints and flows (instead of addressing actors directly) has several advantages. First, it supports fine-grained communication constraints: two actors can communicate only if there are suitable endpoints and flows. It also increases decoupling between senders and receivers, which only operate on their local endpoints, without explicit knowledge of the flows attached to those endpoints. For example, the flow connecting a client to a failed server can be switched over to an alternative server transparently to the client.

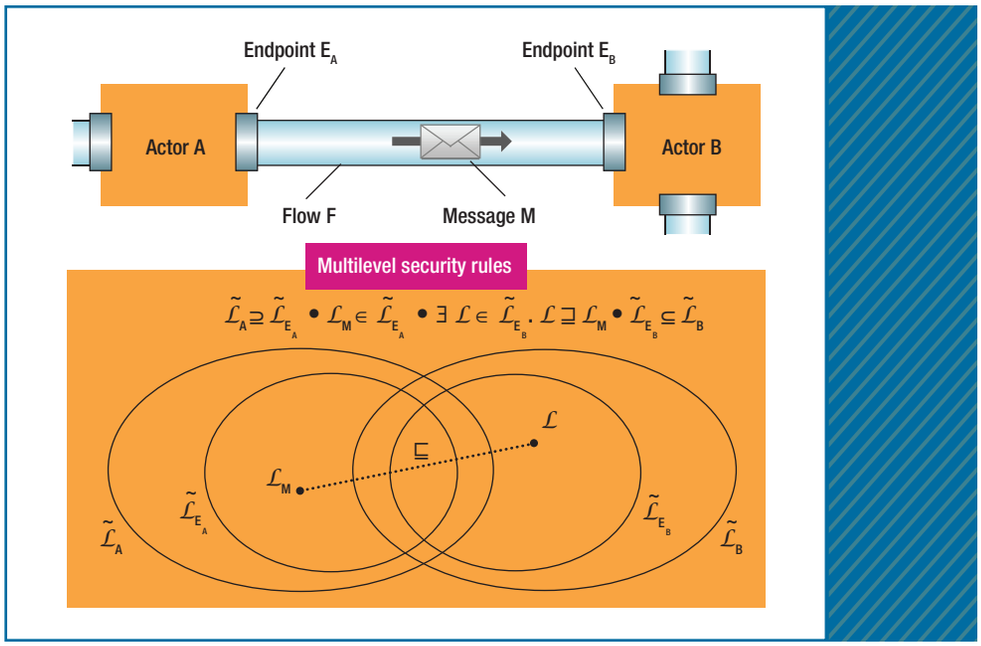


FIGURE 2. Transfer of a message via secure transport. The message goes through a flow that connects an endpoint of the sending actor to an endpoint of the receiving actor. The rules on labels and label sets of actors, endpoints, and messages, guarantee the satisfaction of the Multilevel Security (MLS) policy. The MLS rules are illustrated using Venn diagrams.

Multilevel Security (MLS) Policies

MLS is a well-established concept.[10] It's based on linearly ordered hierarchical classification levels (for example, Unclassified < Confidential < Secret < Top Secret) and nonhierarchical need-to-know categories (for example, mission identifiers). Each organization defines its own levels and categories, in other words, its own labeling domain. In typical systems, which operate in a single labeling domain, a label is a pair LC where L is a level and C is a set of zero or more categories. For example, in the US domain, the label *TS{x,y}* consists of the Top Secret level and identifiers for missions *X* and *Y*.

To support communication among actors from different organizations that can share the common embedded system infrastructure, DREMS uses the novel concept of multidomain labels. A multidomain label has the form $[D_1]L_1C_1 \ldots [D_p]L_pC_p$, where $D_1, \ldots, D_p$ are $p \geq 1$ distinct (identifiers of) domains and each L_iC_i is a label (as defined in single-domain systems) in domain D_i. For example, the label *[US]TS{x}[NATO]CTS{x}* is used for data that is both US Top Secret and NATO Cosmic Top Secret for joint mission X.

The DREMS secure transport security policy follows the standard MLS requirement that information can only flow "up," according to the

dominance relation.[10] For example, a principal with Top Secret clearance can read Unclassified messages, but not vice versa. Data exchanged among different organizations carries labels with levels and categories from all the organizations' domains. Formally, a label $\mathcal{L}$ dominates a label $\mathcal{L}'$, written $\mathcal{L} \sqsupseteq \mathcal{L}'$, if and only if $\mathcal{L}$ has at least all the domains of $\mathcal{L}'$ (and possibly others) and, for each common domain, the level L in $\mathcal{L}$ is at least as high as the level L' in $\mathcal{L}'$ (that is, $L \geq L'$) and the category set C in $\mathcal{L}$ contains the category set C' in $\mathcal{L}'$ (that is, $C \supseteq C'$).

Each actor has an immutable set of labels, which describe the clearance of the actor (what information the actor is allowed to read and write). Only trusted platform actors assign the label set to the actor.

Each endpoint E_A also has an immutable set of labels $\tilde{\mathcal{L}}_{E_A}$, which must be contained in the label set $\tilde{\mathcal{L}}_A$ of the (unique) actor A that owns the endpoint (that is, $\tilde{\mathcal{L}}_{E_A} \subseteq \tilde{\mathcal{L}}_A$). The label set is assigned to the endpoint only by trusted platform actors.

Each message sent via secure transport has an immutable label, which describes the sensitivity of the message. The label is assigned by the actor that creates and sends the message. An actor A can send a message M with label $\mathcal{L}_M$ through an endpoint E_A with label set $\tilde{\mathcal{L}}_{E_A}$ if and only if $\mathcal{L}_M \in \tilde{\mathcal{L}}_{E_A}$.

Figure 2 shows all of these MLS rules. These rules follow the standard MLS policy,[10] adapted to secure transport. When actor A attempts to send message M with label $\mathcal{L}_M$ through endpoint E_A, the secure transport checks that $\mathcal{L}_M \in \tilde{\mathcal{L}}_{E_A}$. When M is received through endpoint E_B of actor B, the secure transport checks that $\mathcal{L} \sqsupseteq \mathcal{L}_M$ for some label $\mathcal{L} \in \tilde{\mathcal{L}}_{E_B}$.

Networks

When a flow connects endpoints on different nodes, secure transport uses IPv6 (http://tools.ietf.org/html/rfc2460) to transfer messages across the network, which may involve various wireless networking devices. Without proper protection, messages traveling through the network could be seen or modified, defeating the MLS policy. IPsec[11] and other measures are used to protect the confidentiality of messages and their labels.

Model-Driven Application Development, Integration, and Deployment

To simplify development and promote producible and verified systems (requirement 7), we have developed a model-based framework for DREMS for developing and integrating applications. This approach uses models to represent the software, the hardware platform, and the mapping between the two. The validation of well-formedness constraints over the models makes the early detection of integration errors possible. Code generators then translate the validated high-level models into low-level artifacts, such as program code and deployment plans to configure the system.

System integration and deployment (requirement 6) are also simplified with this approach. Once individual application models are combined, the global system configuration can be generated the same way as a single application configuration. Global system properties, such as timing, can be checked using the integrated models. The graphical modeling language as a technique, along with reusability via templates in the modeling language, also addresses rapid application development (requirement 1).

Figures 3 summarizes the model-driven development process. During steps 1 and 2, data types are created and used to define the structure and interfaces of individual software components. Multiple implementations of the same component type can coexist, providing the application developer with alternative implementations. Step 3 includes generating skeleton files and using those files to implement the behavior logic of the component. Once a component has been implemented, it can be reused across different applications and projects. Applications are defined by wiring instances of different components together (step 4).

After all applications are modeled, the system integrator performs steps 5 through 7 (described in Figure 3b). Well-formedness (requirement 7) is ensured by a design constraint checker that analyzes the models and reports violations, including details about the constraints violated and the modeling elements involved.

The deployment plan describes all aspects of the application, including the binary libraries required for each component and the metadata describing those libraries, the secure transport configuration, and the component interactions. This plan is provided to the runtime platform's deployment and configuration service that is responsible for deploying and activating the application on the distributed platform (see the example in Figure 4).

An Example

To demonstrate DREMS, we conducted a complex, multinode experiment on a testbed of fanless computing nodes, each containing an Intel Atom N270 clocked at 1.6 GHz and with 1 GByte of RAM. The nodes were connected via a private subnet,

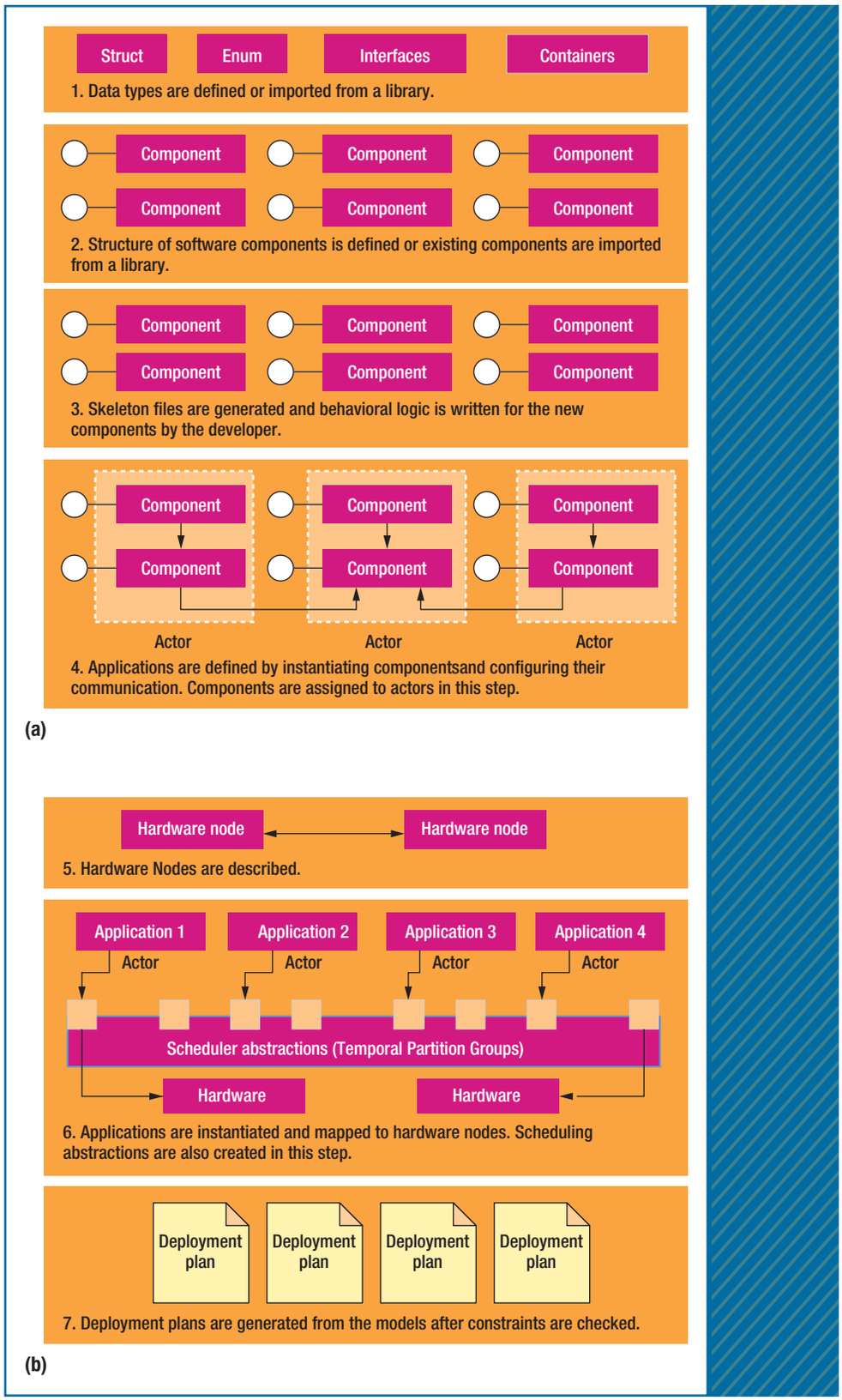


FIGURE 3. The model-driven development process: (a) application development and (b) system integration on a three-node cluster of embedded processors.

which had a network control node running dummynet,[12] allowing full control of the bandwidth, latency, and packet loss on any network link (see Figure 4).

On this testbed, we emulated a cluster of three satellites, each running a copy of an example of a cluster flight control application (CFA). In this example, the CFA consisted of three actors replicated on each satellite: OrbitMaintenance, ModuleProxy, and CommandProxy. OrbitMaintenance keeps track of every satellite's position and updates the cluster with its current position. ModuleProxy connects to the Orbiter space flight simulator,[13] which simulates the satellite hardware and orbital behavior. CommandProxy receives commands from the ground network.

Each node publishes a state vector describing its position and subscribes to the state vectors of all other satellites. Individual state vectors are periodically updated on each satellite through an asynchronous method interface (AMI) from ModuleProxy to OrbitMaintenance. This interaction represents the flight hardware periodically updating the control software with a new satellite state. The connection between the Orbiter and ModuleProxy facilitates periodically getting position data from the satellite sensors.

When OrbitMaintenance receives a command from CommandProxy, it publishes the command as a Satellite_Command topic. The OrbitMaintenance actor on each satellite subscribes to the Satellite_Command topic, and upon reception of the topic, instructs the satellite thrusters to fire (via an

FIGURE 4. The runtime platform's deployment. The simulator image (on the left) shows three satellites, whereas the other display (on the right) shows the deployment model of the experiment.

AMI call to **ModuleProxy**), which activates the satellite thruster in the simulation.

Despite the complexity of the application, only 405 LOC total (0.41 percent of the application code) were written by hand among the four components. The other 99.59 percent is generated code that governs all communications, timing, and interactions.

There certainly exist state-of-the-art development environments and runtime platforms that address some of the needs we've discussed in this article. There are model-based development environments for embedded systems (for example, Mathworks's tool-suites, IBM's UML tools, and so on), there are various real-time operating system products with sophisticated development toolchains (for example, Integrity by Green Hills), and there are systems that support MLS (for example, SELinux). However, to the best of our knowledge, there's no single development environment and runtime platform that holistically provides all of these capabilities in one package.

We believe that emerging cloud paradigms for mobile devices will require the capability to develop, configure, and manage distributed applications and platform services in a manner that enables efficient operation of the platform, permits the use of advanced component models and model-based design for improving modularity and analyzability, and treats information flow security concerns as a first-class concept. We believe the runtime platform and the toolchain we've described in this article will help progress in this direction.

Acknowledgments

The DARPA System F6 Program supported this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of DARPA. We thank Olin Sibert of Oxford Systems and all the team members of our project for their invaluable input and contributions to this effort.

References

1. L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, vol. 54, no. 15, 2010, pp. 2787–2805.
2. O. Brown, P. Eremenko, and C. Roberts, "Cost-Benefit Analysis of a Notional Fractionated SATCOM Architecture," *Proc. 24th Int'l Comm. Satellite Systems Conf.*, American Inst. Aeronautics and Astronautics, 2006; doi:10.2514/6.2006-5328.
3. A. Dubey et al., "A Software Platform for Fractionated Spacecraft," *Proc. IEEE Aerospace Conf.*, IEEE, 2012, pp. 1–20.
4. R.K. Karmani and G. Agha, "Actors," *Encyclopedia of Parallel Computing*, D. Padua, ed., Springer, 2011, pp. 1–11.
5. *Avionics Application Software Standard Interface* (draft 15), document 653, Aeronautical Radio, January 1997.
6. W.R. Otte et al., "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," to be published in *Proc. 16th IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing* (ISORC 13), IEEE.
7. C. Szyperski, "Component Technology: What, Where, and How?," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS, 2003, pp. 684–693.
8. *Light Weight CORBA Component Model* (revised submission), Object Management Group, 2003; www.info.fundp.ac.be/~ven/CIS/OMG/lightweight%20component%20model.pdf.
9. W.R. Otte et al., "Infrastructure for Component-Based DDS Application Development," *Proc. 10th ACM Int'l Conf. Generative Programming and Component Eng.* (GPCE 11), ACM, 2011, pp. 53–62.
10. D.E. Bell and L.J. LaPadula, *Secure Computer Systems: Mathematical Foundations*, tech. report 2547, vol. I, MITRE, 1973.
11. S. Kent and K. Keo, *Security Architecture for the Internet Protocol, IETF RFC 4301*, Internet Society, 2005.
12. M. Carbone and L. Rizzo, "Dummynet Revisited," *SIGCOMM Computing Comm. Rev.*, vol. 40, no. 2, 2010, pp. 12–20.
13. B. Irving, "Playing in Space: Interactive Education with the Orbiter Space Flight Simulator," *Proc. Int'l Space Development Conf.* (ISDC 07), 2007; download.orbit.m6.net/news/PlayingInSpace_ISDC2007_Irving_Slides.pdf.

ABOUT THE AUTHORS

TIHAMER LEVENDOVSZKY is a research assistant professor at Vanderbilt University. His research interests include automated software engineering, model-based engineering, computer security, and performance analysis of software systems. Levendovszky received a PhD in computer science from the Budapest University of Technology and Economics. Contact him at tihamer.levendovszky@vanderbilt.edu.

ABHISHEK DUBEY is a research scientist at Institute for Software-Integrated Systems (ISIS) at Vanderbilt University. His research interests include distributed fault-tolerant real-time systems and autonomic computing. Dubey received a PhD in electrical engineering from Vanderbilt University. Contact him at dabhishe@isis.vanderbilt.edu.

WILLIAM R. OTTE is a research scientist at ISIS at Vanderbilt University. His research interests include middleware for real-time embedded systems and their deployment and configuration. Otte received a PhD in computer science from Vanderbilt University. Contact him at wotte@dre.vanderbilt.edu.

DANIEL BALASUBRAMANIAN is a research scientist at ISIS at Vanderbilt University. His research interests include the lightweight application of formal methods and analysis to model-based development. Balasubramanian received a PhD in computer science from Vanderbilt University. Contact him at daniel@isis.vanderbilt.edu.

ALESSANDRO COGLIO is a principal scientist at Kestrel Institute. His research interests include formal methods and tools to develop correct-by-construction software via formal specification, refinement, and theorem proving. Coglio received an MS in informatics engineering from University of Genoa. Contact him at coglio@kestrel.edu.

SANDOR NYAKO is a senior research engineer at Vanderbilt University. His research interests include the telecom, finance, and computer entertainment fields. Nyako received a BSc in computer science from Eotvos Lorand University. Contact him at snyako@isis.vanderbilt.edu.

WILLIAM EMFINGER is a graduate research assistant at ISIS at Vanderbilt University. His research interests include networking for critical systems. Emfinger received a BE in electrical engineering and biomedical engineering from Vanderbilt University. Contact him at emfinger@isis.vanderbilt.edu.

PRANAV SRINIVAS Kumar is a graduate research assistant at ISIS at Vanderbilt University. His research interests include modeling, analysis, and verification techniques for distributed component-based software applications. Kumar received a BE in electronics and communications engineering from Anna University. Contact him at pkumar@isis.vanderbilt.edu.

ANIRUDDHA GOKHALE is an associate professor in the department of electrical engineering and computer science and senior research scientist at ISIS at Vanderbilt University. His research interests include deployment and configuration and quality of service issues, such as timeliness, fault tolerance, and security, in large-scale, service-oriented, distributed, real-time, and embedded systems. Gokhale received a PhD in computer science from Washington University, St. Louis. He's a senior member of IEEE and ACM. Contact him at a.gokhale@vanderbilt.edu.

GABOR KARSAI is a professor of electrical and computer engineering and computer science at Vanderbilt University and senior research scientist at ISIS. His research interests include model-integrated computing (MIC), design automation for model-driven development processes, automatic program synthesis, and the application of MIC in various government and industrial projects. Karsai received a PhD in electrical engineering from Vanderbilt University. He's a senior member of the IEEE Computer Society. Contact him at gabor.karsai@vanderbilt.edu.