Evaluating COTS Component Dependability in Context

Paolo Donzelli, *University of Maryland*

Marvin Zelkowitz and Victor Basili, *University of Maryland and Fraunhofer Center for Experimental Software Engineering*

Dan Allard and Kenneth N. Meyer, *NASA Jet Propulsion Laboratory*

Selecting a COTS component to integrate into an application can be risky. To reduce the risk, this method empirically evaluates component dependability for specific application contexts.

Over the past decade, the software industry has increasingly expanded its adoption of COTS components[1] for complex, mission-critical applications. Using COTS products can shorten development and deployment time because they let system developers focus on creating domain-specific services.[2] Selecting the right COTS component, however, is no easy task.[3] The emerging system's dependability properties—including reliability, security, and availability—result from the chosen components, and failures may arise because they do not behave as anticipated. To make the right choice from among what are often functionally equivalent alternatives, system developers must have empirical evidence that clearly shows a component's dependability.

The International Federation for Information Processing[4] defines dependability as "the trustworthiness of a computing system that allows reliance to be justifiably placed on the services it delivers." However, "reliance" is contextually subjective and depends on the particular stakeholders' needs. Different stakeholders will focus on different system attributes—such as availability, catastrophic-failure avoidance, and deliberate-intrusion prevention—as well as require different levels of adherence to such attributes. The same attribute can also mean different things to different people, and multiple definitions of the same attribute are common.[5,6] Dependability clearly assumes a precise meaning only when applied in a specific context.

From this perspective, we present a practical process that developers can use to empirically evaluate component dependability in their context. Our approach uses the Unified Model of Dependability,[7] a requirements engineering approach specially devised to capture dependability in context. As the "Unified Model of Dependability" sidebar explains, UMD transforms the system developers' high-level dependability needs into a detailed dependability model of the component. This model clearly specifies the measurable characteristics the component must have to be dependable in a specific context. The model then serves as a reference, providing guidance on effectively designing experiments to compare similar components and interpret collected

 0740-7459/05/$20.00 © 2005 IEEE

The Unified Model of Dependability

The UMD requirements engineering framework helps developers elicit and model dependability requirements. UMD is failure-centered in that it permits stakeholders to express their requirements by specifying what they see as the actual *failure*, or class of failures, that should not affect the system or a specific service (*scope*). For each failure, stakeholders can also specify the tolerable manifestations (*measure*) and the desired corresponding system *reaction*. Stakeholders might also specify external events that could harm the system. Figure A shows UMD's basic modeling concepts: failure, scope, measure, reaction, and event. Stakeholders use these concepts to express their dependability requirements. For example, for an RT JVM, a requirement expressed using UMD could be, "The thread scheduler [scope] should not have a jitter bigger than 500 ns [failure] more often than 0.1 percent of the cases [measure]; if the failure occurs, a warning should be issued [reaction]."

UMD is stakeholder oriented, as failure, scope, measure, reaction, and event are basic concepts that stakeholders can easily grasp and associate with entities in their application domain. Rather than dealing with abstract entities such as dependability and its attributes, stakeholders can use UMD to focus on practical concepts and more effectively map their dependability needs to their context. Moreover, to better support stakeholders in formulating their requirements and to address the needs of a particular application context, UMD lets users refine its basic modeling concepts. As figure A shows, for example, to help stakeholders identify unacceptable system or service failures, it might suggest the different types of failures that could occur (response time, timeliness, and so on).

To implement UMD, we developed a Web-enabled tool[1,2] organized around two main tables:

- The *scope table*, which allows stakeholders to define all dependability-related services starting from the system functional description.
- The *failure table*, which allows stakeholders to specify their requirements by defining, for the whole system or a specific service, undesired failures, their tolerable manifestations, possible triggering external events, and desired reactions.

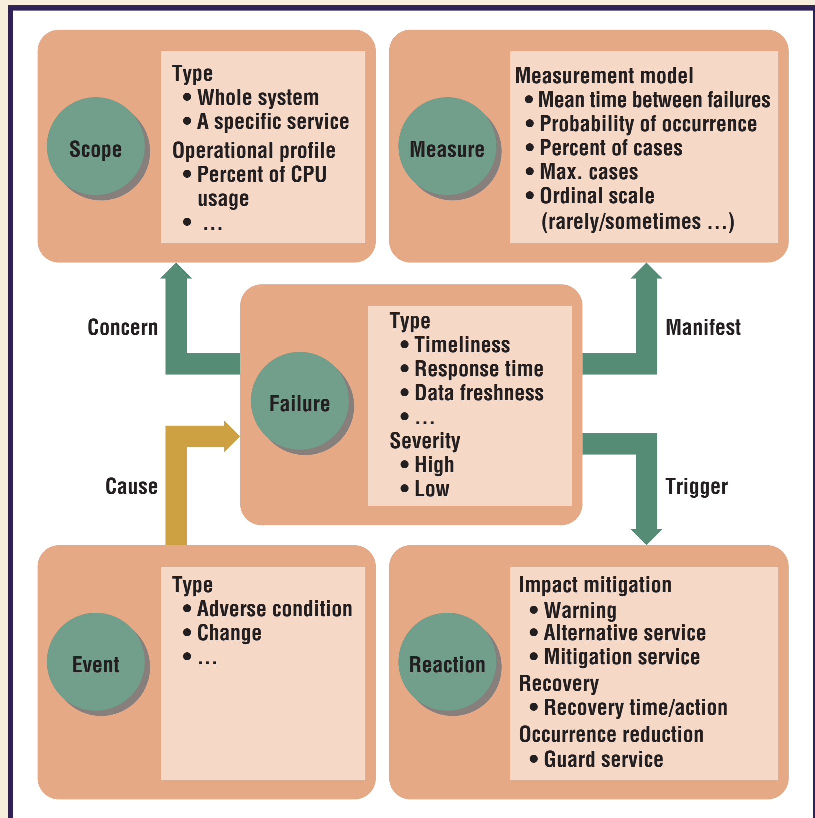

Figure A. The Unified Model of Dependability's concepts and relationships. Stakeholders express dependability requirements using the concepts of failure, scope, system reaction, and potentially threatening external events.

Reference

1. V. Basili, P. Donzelli, and S. Asgari, "A Unified Model of Dependability: Capturing Dependability in Context," *IEEE Software*, vol. 21, no. 6, 2004, pp. 19–25.
2. P. Donzelli and V. Basili, "A Practical Framework for Eliciting and Modeling System Dependability Requirements: Experience from the NASA High Dependability Computing Project," to appear in *J. Systems and Software*, 2005.

data. The process can be applied to any specific context and COTS component. Here, we describe our comparison of the dependability of real-time Java virtual machines (RT JVM) in the spacecraft software context.

The real-time Java virtual machine

Today, the Java programming language helps many companies reduce development costs and achieve faster time-to-market. Java technology has widely penetrated the general-

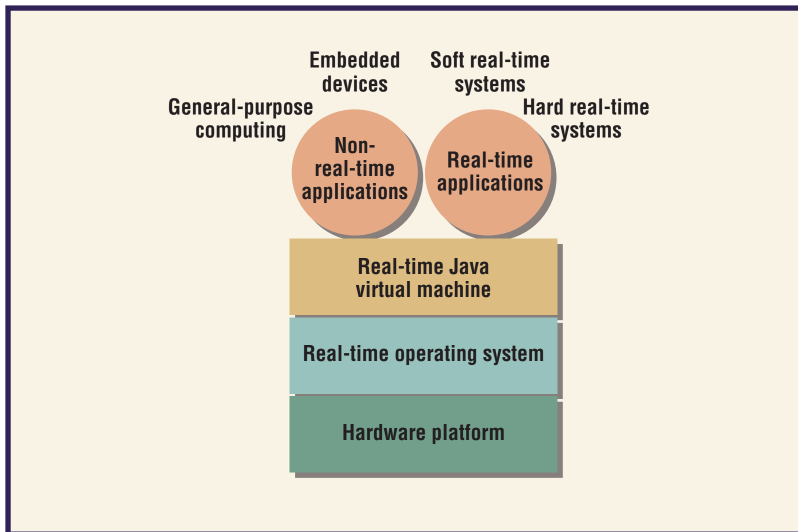

Figure 1. The real-time Java virtual machine as a component.

purpose computing and embedded device markets, where manufacturers exploit its benefits. However, in the real-time arena, Java lags behind. Although early results of use are promising, Java's adoption for critical applications is an open issue.[8]

The Java technology model with real-time extensions, as specified in the Real-Time Specification for Java (RTSJ),[9] leverages real-time operating system (RTOS) capabilities by combining Java's benefits with the possibility of correctly reasoning about the executing software's temporal behavior. Java addresses the requirements faced by real-time and non-real-time applications by coupling the RT JVM with an RTOS and providing developers mechanisms to separate hard real-time and soft real-time threads, objects, and memory.

As figure 1 shows, the RT JVM is the real-time Java technology model's key component. Thus, whereas the JVM for standard non-real-time applications is typically considered an implicit choice and sometimes even a commodity,[10] the RT JVM is a COTS component that can greatly impact a project's shape.

Currently, several RTSJ-compliant RT JVM implementations are available.[11] The selection (and subsequent adoption) of a specific RT JVM requires a careful trade-off analysis involving economical benefits and potential risks. The benefits of adopting an RT JVM are counterbalanced by the risk of applying it for real-time applications, rather than other more traditional and dependable technologies. Understanding to what extent we can depend on a specific RT JVM is crucial.

Case study: RT JVM for mission-critical spacecraft software

Spacecraft software is a classical hard real-time application.[11,12] The issues are complex, involving numerous sensors, control mechanisms, and actuators. High performance is insufficient; the software's temporal behavior must be predictable. The software requires that certain threads execute at certain times, depending on the type of activity to be performed (such as periodic control loops or exceptional events handlers).

Current spacecraft software uses more traditional languages—typically C and C++.[11] NASA's Jet Propulsion Laboratory is actively investigating the possibility of adopting Java for mission-critical spacecraft software in projects such as Golden Gate.[11,12] The idea is to make systems coalesce and have both real-time and non-real-time code on the same JVM so that both ground systems and spacecraft can have the same processors and environments.

Our goal is to evaluate some of the available RT JVMs to understand their dependability for spacecraft software. Because any technology has strengths and weaknesses that vary in relevance depending on application context,[3] our evaluation process aims to provide empirical observations that can help developers understand individual RT JVM characteristics for the spacecraft software context. During the case study, we evaluated three RT JVMs: *VM-A* and *VM-B*, running on the same RTOS and hardware platform, and *VM-C*, running on a different RTOS and platform.

The empirical evaluation process

As figure 2 shows, our evaluation process consists of three main steps.

We first build a dependability model of the component. To do this, we identify the system developers' dependability needs and transform them into measurable characteristics that the component must possess to be dependable for the specific context. The dependability model sets the dimensions of our *evaluation space*: the characteristics we'll use to compare the candidate components. For example, we could compare RT JVMs based on jitter—the difference between the desired and actual scheduled time for a real-time thread.[9] For a dimension, we might also set a precise value that acts as a gate during the evaluation process, and only consider components that meet the precise re-

quirement. For example, for an RT JVM we could say that jitter must not exceed 50 μs. As we describe below, to build the dependability model, we adopt UMD, a requirements engineering approach especially devised to capture dependability in context.

Next, we design and implement the experiments. To design experiments suitable for measuring and identifying component characteristics, we use the information we acquired in building the dependability model. By precisely specifying the information we need to collect, the dependability model supports the identification of the areas that our experiments should focus on.

Finally, we interpret the results to evaluate the candidate components' dependability. Our interpretation is facilitated by the dependability model, which precisely defines the evaluation space dimensions (that is, the relevant component characteristics and evaluation gates).

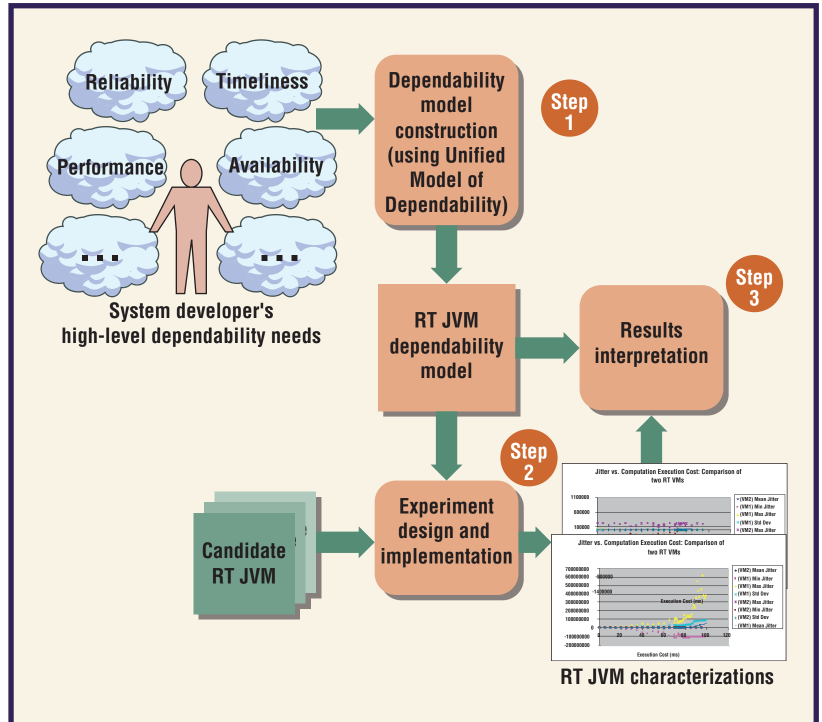

Figure 2. Dependability evaluation. We first construct a dependability model, then use the model's information to design, run, and interpret experiments on component dependability.

Dependability model construction

Using UMD, system developers specified characteristics that an RT JVM should possess to be considered dependable for spacecraft software. They used the UMD tool to build the RT JVM dependability model in two steps.

First, by analyzing descriptions of the candidate RT JVMs, they identified services for which they believed dependability was relevant. Figure 3 shows an extract from the resulting scope table.

Second, guided by UMD's failure table structure, they filled as many tables as necessary to transform their dependability needs into precisely specified component characteristics. The refinement of UMD's scope, failure, event, measure, and reaction concepts guided system developers while building the model. Examples of the adopted failure types include

- *Functional correctness*: Functional requirements aren't implemented.
- *Throughput*: The number of items (such as threads) per unit of time is less than expected.
- *Response time*: Response time is greater than expected.
- *Load*: The number of items handled by the system or service is less than expected.
- *Timeliness*: Jitter, latency, or delay is bigger than expected.
- *Accuracy*: Data accuracy is lower than expected.
- *Data freshness*: Frequency of data updating is less than expected.

Figure 4 shows a failure table example. The system developers signal a failure for the service "Thread Monitor" when a deadline is missed (that is, when a real-time thread is not com-

Start | Create/Delete Model | Select Model | User Registration | Login/Logout

SCOPE: Identification of the relavant services

Name	Description	Delete
System	Real-Time Java Virtual Machine	
Thread Scheduler	Schedule a thread when required (period, exception, etc)	
Thread Monitor	Ensure a thread to be completed within its deadline	
Deadline handler	Signal a missed deadline	
Cost overrun handler	Signal actual cost greater than extimated one	

Figure 3. A sample scope table for the real-time Java virtual machine (RT JVM). Using this table, spacecraft system developers identified potentially critical services.

Figure 4. A sample failure table for the RT JVM. This table helps stakeholders use UMD's basic concepts to specify requirements.

Scope	Event	Failure	Measure	System Reaction
Select from scope list: Thread Monitor	Description: N/A	Description: Missed Deadline	Measure Type and Value: Max cases, 0	Warning Services: Deadline handler (ADD)
CPU Usage: Any	Event Type: N/A	Severity: High		Alternative Services: (ADD)
				Mitigation Services: (ADD)
				Guard Services: (ADD)
				Recovery Behavior: MTTR and MaxTTR, Mean: Max:
	Notes:	Notes:	Notes: No deadline must be missed	Notes.

pleted within the expected time[9]). This is a functional correctness and high severity failure. System developers deem this failure inadmissible for spacecraft applications, and specify that it should never happen (Max cases = 0). However, to increase their confidence in the component, they ask for a warning service (deadline handler) in case a deadline is missed[9] so that applications can take necessary emergency actions.

Figure 5 shows an excerpt from the final dependability model: Each line of the model contains information gathered through the UMD tool's failure table (for example, line 4 corresponds to figure 5's failure table).

Line 1 specifies the system developers' requirement that an RT JVM always be able to handle at least 90 real-time threads. This is a precise requirement from the spacecraft software domain: It sets a *gate* that candidate RT JVMs will have to pass during the evaluation.

Line 2 specifies that the RT JVM be able to detect and react to a "priority inversion."[8] A priority inversion happens when, because of an external application's error (external event), a higher-priority thread cannot access a shared resource locked by a (dormant) lower-priority thread, preempted by a third thread, whose priority is between the priority levels of the other two. In this case, the RT JVM must be able to detect the situation and operate a "priority inheritance," raising the dormant thread's priority to the priority level of the higher-priority thread so that it can be scheduled and free the resource. Again, this is a precise requirement—that is, an evaluation gate.

Line 3 signals that high jitter values could prevent an RT JVM from being adopted for spacecraft applications. In this case, system developers decided not to set a precise value, but to leave the jitter as a *to-be-evaluated* (TBE) characteristic. Jitter therefore sets an evaluation space dimension—that is, a characteristic against which developers will compare the candidate components. As a further comparison element, system developers decided to annotate the *current technology capability* (CTC)—that is, the jitter provided by schedulers currently used for spacecraft software.

Finally, lines 5 and 6 indicate the relevance of the context switch time (the time required to switch context from one thread to another) and the time necessary to throw the deadline handler if a deadline is missed. Again, system developers chose not to specify a precise value for the context switch time, leaving it as a TBE quantity. They did precisely specify that a deadline handler must be thrown within a given percentage of the context switch time (150 percent), setting an evaluation gate. As this example shows, while building a dependability model it's possible to explicitly express links between component characteristics.

The final dependability model (partially illustrated in figure 5) consists of 24 lines and was developed in about 40 hours. This effort was led by an UMD expert (from the University of Maryland) and a JPL specialist, with various interactions from other NASA experts on spacecraft software.

Experiment design

We derived the set of experiments for evaluating the candidates directly from the dependability model in figure 5. First, we identi-

Line	Scope	Event	Failure	Measure	Reaction
1	System CPU usage: Any	N/A	"Handles fewer than 90 real-time threads" Type: Load Severity: High	Max cases = 0 (gate)	N/A
2	Thread scheduler CPU usage: N/A	Priority inversion (application error)	"Unable to detect that a higher-priority thread cannot access a shared resource owned by a dormant lower-priority thread" Type: Functional correctness Severity: High	Max cases = 0 (gate)	Mitigation service: Priority inheritance
3	Thread scheduler CPU usage: Any (with 90 threads)	N/A	"Mean jitter bigger than **TBE**" **CTC**: Jitter := ~500 ns Type: Timeliness Severity: High	Standard deviation = **TBE**	N/A
4	Task monitor CPU usage: Any (with 90 threads)	N/A	"Missed deadline" Type: Functional correctness Severity: High	Max cases = 0 (gate)	Warning: Deadline handler
5	Thread scheduler CPU usage: Any (with 90 threads)	N/A	"Thread context switch time greater than **TBE**" **CTC**: Context switch time = ~1 μs Type: Functional correctness Severity: High	Probability of occurrence = **TBE**	N/A
6	Deadline handler CPU usage: Any (with 90 threads)	N/A	"Time to throw the deadline handler is greater than 150 percent of the thread context switch time" Type: Response time Severity: High	Max cases = 0 (gate)	N/A

Evaluation gate
To-be-evaluated characteristic
Current technology capability
Link

Figure 5. Excerpt from the real-time Java virtual machine dependability model for the spacecraft software domain.

fied the actions required to verify each line of the dependability model (for example, for line 1 in figure 5, verification that a candidate component could handle at least 90 real-time threads). We then clustered these actions to minimize the number of required experiments. For the sake of brevity, we describe only two of these experiments here:

- *Experiment 1: Priority inversion*. We ran three real-time threads on each candidate RT JVM with three different priorities—high, medium, and low—to generate a priority inversion. This experiment covers line 2 of the dependability model.
- *Experiment 2: Workload of 90 increasingly expensive real-time threads*. On the candidate RT JVMs, we ran 10 workload sets of 90 threads (sets S1 to S10). All threads within a set have the same cost (that is, the same computational CPU time), which increases regularly from set to set (from 1 ms for S1 to 10 ms for S10). This experiment covers lines 1, 2, and 4 of the dependability model.

To prepare for the experiments, a JPL software specialist developed about 17,000 lines of Java code, requiring about 400 hours, including 100 hours for running the experiments.

Results interpretation

Table 1 summarizes the results from our experiments. Each row in the table corresponds to a line of the dependability model (figure 5), indicating the corresponding exper-

Table 1

Excerpt from the experimental results

Dependability model correspondence	Experiment number	Results VM-A	VM-B	VM-C
Line 1 - 90 real-time threads	2	Pass	Pass	**Fail**
Line 2 - Priority inversion	1	Pass	Pass	Pass
Line 3 - Jitter	2	Figure 6	Figure 6	N/A
Line 4 - Missed deadline	2	Figure 6	Figure 6	N/A

iment and obtained results. For example, row 2 of table 1 indicates that experiment 1 evaluated the RT JVMs' ability to detect a priority inversion (line 2 of the dependability model)—an experiment that all three RT JVMs passed.

In experiment 2, we noticed that VM-C couldn't run 90 real-time threads (see table 1, row 1). Given this, we focused on VM-A and VM-B. Figure 6 shows the results for jitter mean values and corresponding standard deviations for both VM-A and VM-B for each of the 10 workloads. The green arrows indicate the first workload in which the RT JVMs missed a deadline. As our results show, VM-B

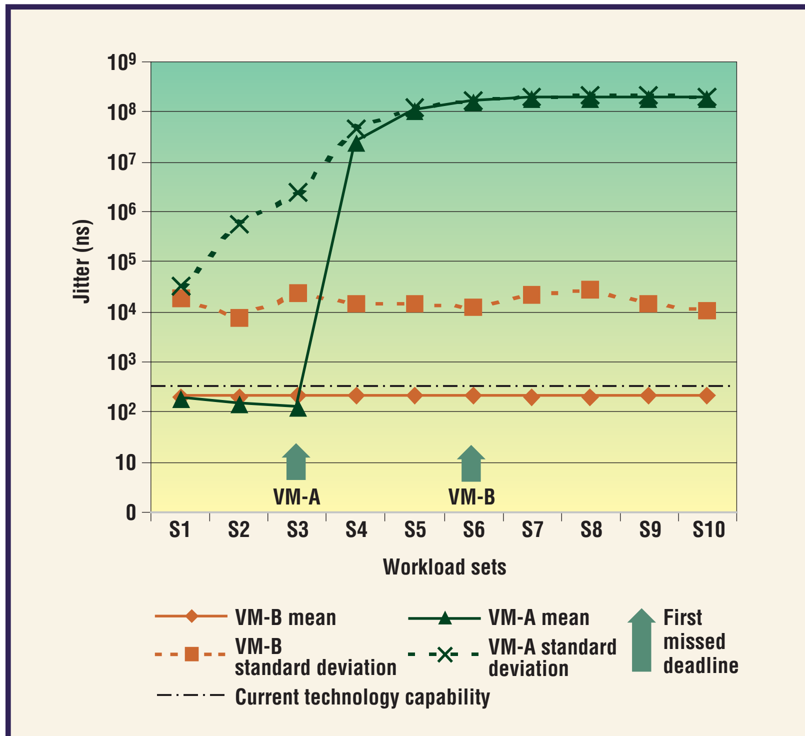

Figure 6. Results from experiment 2, which tested 10 workload sets of 90 threads. Each set exacted an increasing computational cost (S1 = 1 ms … S10 = 10 ms).

both provides a lower jitter (in terms of mean and standard deviation) than VM-A and meets its deadlines for considerably higher workloads. As figure 6 also shows, VM-B's mean jitter is always lower than the CTC.

Our results provide a clear picture of the candidates' behavior in our context. In addition to seeing how each candidate RT JVM behaved relative to the spacecraft software domain—which is crucial for the selection process—we also clearly identified RT JVM weaknesses that we must address before proceeding with the integration. In particular, some unexpected RT JVM behavior has led us to collaborate with vendors to improve the products either by enhancing nonfunctional characteristics (such as being able to handle more than 90 real-time threads) or by adding desired functionalities (such as a deadline handler) to better cope with potential failures.

The case study results have increased our confidence in the process suggested for evaluating COTS components dependability. The process produced a clear, quantitative characterization of the candidate components in line with the high-level dependability properties that system developers sought.

The process let us identify individual component's strengths and weaknesses in the target context. This knowledge not only supports the selection process in achieving a better fit between the components and their intended use, but also identifies the weaknesses that we must address to improve their dependability. In particular, on the basis of the experimental results, we could decide how components must be changed to achieve behavior appropriate to the context (as in our case study) or to adopt integration mechanisms suitable to prevent or mask undesired behavior. For example, it might be possible to adopt "wrapper software" to filter dangerous external events (as in priority inversion, line 2, figure 5), or augment components with desired reactions (not originally designed in the component but captured through UMD as dependability requirements) to potential failures.

Conducting context-focused, UMD-derived experiments complements vendor component evaluations by placing the component in the

user's context and focusing on specific aspects that the component's creator might have neglected. From this perspective, we address the classical limits of empirical evaluation in two ways:

- We narrow the experiment coverage problem by using UMD to perform a careful and systematic elicitation of the context dependability needs.
- We identify highly focused experiments, measuring and stimulating detailed components characteristics.

Developers can adopt UMD to identify dependability requirements for systems in any context[7] and apply our evaluation process whenever they need to understand a component's dependability in context. Our future work will focus on further evaluating our suggested process with more general-purpose COTS products.

Acknowledgments

The NASA High Dependability Computing Program supported our work under cooperative agreement NCC-2-1298.We thank Jennifer Dix for proofreading this article.

References

1. V. Basili and B. Boehm, "COTS-Based Systems Top 10 List," *Computer*, vol. 34, no. 5, 2001, pp. 91–93.
2. J. Voas, "COTS Software: The Economical Choice?" *IEEE Software*, vol. 15, no. 2, 1998, pp. 16–19.
3. I. Gorton, A. Liu, and P. Brebner, "Rigorous Evaluation of COTS Middleware Technology," *Computer*, vol. 36, no. 3, 2003, pp. 50–55.
4. International Federation for Information Processing Working Group 10.4, www.dependability.org.
5. B. Boehm et al., "The ROI of Software Dependability: The iDAVE Model," *IEEE Software*, vol. 21, no. 3, 2004, pp. 54–61.
6. J.C. Laprie, ed., *Dependability: Basic Concepts and Terminology—Dependable Computing and Fault Tolerance*, vol. 5, Springer-Verlag, 1992.
7. V. Basili, P. Donzelli, and S. Asgari, "A Unified Model of Dependability: Capturing Dependability in Context," *IEEE Software*, vol. 21, no. 6, 2004, pp. 19–25.
8. B. Sanden, "Coping with Java Threads," *Computer*, vol. 37, no. 4, 2004, pp. 20–27.
9. G. Bollella et al., *The Real-Time Specification for Java*, Addison Wesley, 2000.
10. M. Torchiano and M. Morisio, "Overlooked Aspects of COTS-Based Development," *IEEE Software*, vol. 21, no. 2, 2004, pp. 88–93.
11. E.G. Benowitz and A.F. Niessner, "Experiences in Adopting Real-Time Java for Flight-Like Software," *Proc. 4th Quality Mission Software Workshop*, Springer Verlag, 2002, pp. 490–496.
12. D. Dvorak et al., "Project Golden Gate: Towards Real-Time Java in Space Missions," *Proc. IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing*, IEEE CS Press, 2004, pp. 15–22.

About the Authors

Paolo Donzelli is a visiting senior research scientist in the University of Maryland's Computer Science Department, on leave from Italy's Office of the Prime Minister, where he's a director with the Department of Innovation and Technology. His research interests include software process improvement, requirements engineering, and dependability modeling and validation. He received his PhD from the University of Rome TorVergata. Contact him at the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; donzelli@cs.umd.edu.

Marvin Zelkowitz is a professor in the University of Maryland's Computer Science Department and Institute for Advanced Computer Studies, and chief scientist at the Fraunhofer Center, Maryland. He received his PhD in computer science from Cornell University. He's an IEEE Fellow. Contact him at the Dept. of Computer Science and Inst. for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742; marv@zelkowitz.com.

Victor Basili is a professor of computer science at the University of Maryland, College Park, where he works on measuring, evaluating, and improving the software development process and product. He received his PhD in computer science from the University of Texas. He's an IEEE and ACM Fellow. Contact him at the Dept. of Computer Science and Inst. for Advanced Computer Studies, Univ. of Maryland, College Park, MD 20742; basili@cs.umd.edu.

Dan Allard is a senior software systems architect at the Jet Propulsion Laboratory in Pasadena, California. His research interests include disruption-tolerant distributed information systems, software dependability, agent frameworks, and applications of real-time Java technology to JPL space software systems. He received his BS in engineering physics at Tufts University. Contact him at the Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, CA 91109; dan@jpl.nasa.gov.

Kenneth N. Meyer works at the Jet Propulsion Laboratory in Pasadena, California, where he's task manager for the Mission Data System-05 project and MDS Outreach Lead for the Information Technologies Program Office. Contact him at the Jet Propulsion Laboratory, 4800 Oak Grove Dr., Pasadena, CA 91109; kenneth.n.meyer@jpl.nasa.gov.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.