OASIS: A Service-Oriented Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems

James H. Hill
Indiana University-Purdue University Indianapolis
Indianapolis, IN USA
hillj@cs.iupui.edu

Hunt Sutherland
GE Global Research
Niskayuna, NY USA
sutherland@crd.ge.com

Paul Staudinger
GE Global Research
Niskayuna, NY USA
staudinger@crd.ge.com

Thomas Silveria
Raytheon Company
Portsmouth, RI USA
thomas_silveria@raytheon.com

Douglas C. Schmidt
Vanderbilt University
Nashville, TN USA
d.schmidt@vanderbilt.edu

John M. Slaby
Raytheon Company
Portsmouth, RI USA
john_m_slaby@raytheon.com

Nikita A. Visnevski
GE Global Research
Niskayuna, NY USA
visnevsk@research.ge.com

Abstract

Performance analysis tools for enterprise distributed real-time and embedded (DRE) systems require instrumenting heterogeneous sources (such as application- and system-level hardware and software resources). Traditional techniques for software instrumentation of such systems, however, are tightly coupled to system design and metrics of interest. It is therefore hard for system testers to increase their knowledge base and analytical capabilities for enterprise DRE system performance using existing instrumentation techniques when metrics of interest are not known during initial system design.

This paper provides two contributions to research on software instrumentation for enterprise DRE systems. First, it presents OASIS, which is service-oriented middleware for instrumenting enterprise DRE systems to collect and extract metrics without design time knowledge of which metrics are collected. Second, this paper empirically evaluates OASIS in the context of a representative enterprise DRE system from the domain of shipboard computing. Results from applying OASIS to a representative enterprise DRE system show that its flexibility enables DRE system testers to precisely control the overhead incurred via instrumentation.

1. Introduction

Emerging trends and challenges. Enterprise distributed real-time and embedded (DRE) systems (such as power grid management, shipboard computing environments, and traffic management systems) are often composed of loosely coupled applications that run atop distributed infrastructure software (*e.g.*, service-oriented middleware [10]) and deployed into heterogeneous execution environments. In addition to meeting their functional requirements, these types of DRE systems must also meet quality-of-service (QoS) requirements, such as latency, response time, and scalability, so that applications achieve their expected behavior in a timely manner [13].

To ensure QoS requirements, DRE system testers must analyze system performance via runtime monitoring of systems behavior and resources (*e.g.*, CPU usage, memory allocation, and event arrival rate). Conventional approaches for monitoring enterprise DRE system behavior and resources employ software instrumentation techniques [2, 8, 11, 13] that collect metrics of interest (*e.g.*, busy time for an application, L2 cache memory usage, state of a component, and number of heartbeats sent by a watchdog daemon) as a system runs in its target environment. Performance analysis tools (such as testing and experimentation, distributed resource management, and real-time monitoring/reporting) can then be used to evaluate collected metrics and inform DRE system testers if the system meets its QoS requirements. These tools can also identify bottlenecks in system and application components that exhibit high and/or unpredictable behavior and resource usage [9].

Although software instrumentation enables performance analysis tools to evaluate enterprise DRE system QoS requirements, conventional techniques [13] for collecting metrics are tightly coupled to system implementations, *i.e.*, DRE system developers must decide what metrics to collect and then incorporate the neces-

sary probes to collect these metrics *during the system design phase*. The drawback with this approach, however, is that developers must either (1) redesign their systems to incorporate the new/different metrics or (2) use *ad hoc* techniques, such as augmenting existing code to inject the instrumentation hooks, without understanding the impact on overall system design and maintainability. DRE system developers therefore need better techniques to simplify instrumenting enterprise DRE systems to collect and extract metrics, especially when the desired metrics are not known *a priori*.

Solution approach → Service-oriented dynamic software instrumentation. Dynamic software instrumentation [2, 12] is a technique that enables developers to control at runtime what and how metrics are collected, as opposed to making such decisions during the design phase. When dynamic software instrumentation is combined with service-oriented middleware methodologies, dynamic instrumentation no longer need be a concern of DRE system developers. Instead, it becomes a service that collects metrics from the system and prepares the metrics for evaluation by performance analysis tools. Moreover, this process can occur without *a priori* knowledge (*e.g.*, structure and quantity) of metrics being collected.

This paper describes the *Open-source Architecture for Software Instrumentation of Systems* (OASIS), which is a service-oriented dynamic instrumentation middleware for enterprise DRE systems. OASIS enables DRE system developers and testers to collect metrics from a distributed environment at runtime without making such decisions on how to do so at design time. This loose coupling enables performance analysis tools to leverage OASIS's services to analyze collected metrics, such as querying for specific response times of events. We also present the results of experiments that apply OASIS to a representative enterprise DRE system. These results show that OASIS enables DRE system developers to dynamically configure the instrumentation middleware so that end-to-end response time is not impacted significantly.

Paper organization. The remainder of this paper is organized as follows: Section 2 motivates OASIS via a case study from the domain of shipboard computing; Section 3 describes the structure and functionality of OASIS; Section 4 presents empirical results that evaluate OASIS's impact on representative QoS requirements; Section 5 compares work on OASIS with related R&D efforts; and Section 6 presents concluding remarks.

2. Case Study: The SPRING Scenario

This section motivates the need for OASIS via a case study from the domain of shipboard computing. Figure 1 shows the SPRING scenario, which is a representative enterprise DRE system from the domain of shipboard computing. This scenario tests QoS concerns (such as throughput,

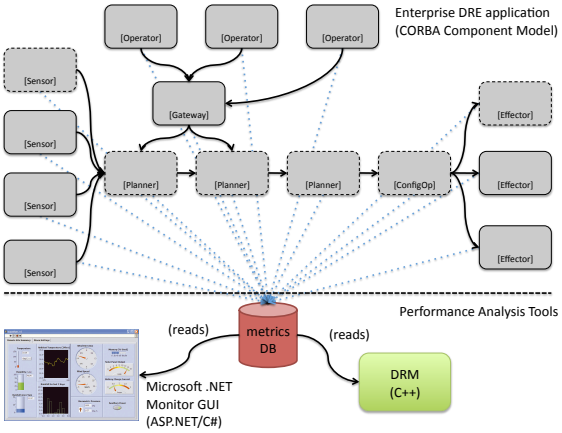

Figure 1. Overview of the SPRING Scenario

latency, jitter, and scalability) of the underlying enterprise DRE system infrastructure (*e.g.*, hardware, middleware, operating system, and network) that will host it. For example, the scenario evaluates whether the infrastructure middleware can manage and meet the resource needs of applications in volatile conditions, such as changing arrival rates of events and dynamic resource usage. Detailed coverage of the SPRING scenario appears in [4].

Existing techniques for instrumenting enterprise DRE systems, such as the SPRING scenario, assume software instrumentation concerns (*e.g.*, what metrics to collect and how to extract metrics from the enterprise DRE application) are incorporated into the system's design. Tightly coupling system instrumentation with the existing software design, however, is not ideal because it prevents flexibility in the system's design and implementation as its evolves throughout the software lifecycle, such as modifying the number of components and substituting different technologies for validating QoS requirements. In general, developers and testers of the SPRING scenario face the following challenges:

Challenge 1: Dynamic collection and extraction of metrics without design time knowledge metric composition. Since the SPRING scenario is under development it is hard for DRE system developers and testers to know all the metrics they need to collect for performance analysis tools during early phases of the software lifecycle. Moreover, the SPRING scenario has many operating conditions (such as high and low event throughput) and the QoS of its critical path can be affected by instrumentation. DRE system developers and testers therefore need mechanisms that enable them to collect and extract metrics without making such de-

cisions at design time, and dynamically modify collection to ensure instrumentation does not unduly perturb performance. Section 3.2.1 shows how OASIS addresses this challenge by using a multi-level packaging technique that enables the it to adapt to different metrics being collected.

Challenge 2: Dynamic discovery of metrics for analysis without design time knowledge of metric composition. Existing third-generation languages and distribution middleware technologies, such as Java, CORBA, and Microsoft .NET, support dynamic types that enable transmission of data without knowledge of its structure and quantify. Although dynamic types are a plausible solution for this challenge, they are tightly coupled to a particular programming languages and middleware. Since many enterprise DRE systems (including the SPRING scenario) are composed of heterogenous technologies DRE system developers and testers need higher-level mechanisms that enable them to discover metrics for performance analysis tools without being bounded to specific technologies. Section 3.2.2 shows how OASIS addresses this challenge by using metadata to describe metrics being collected beforehand so it is possible to discover the actual metrics later.

These challenges make it hard for DRE system developers and testers to apply different performance analysis tools to the SPRING scenario throughout the software lifecycle, particularly in early phases. Moreover, these challenges extend beyond the SPRING scenario and apply to other enterprise DRE systems that need to collect and extract metrics without knowledge of such concerns at design time.

3. The Structure and Functionality of OASIS

This section describes OASIS and explains how it addresses the design challenges presented in Section 2 that DRE system developers and testers encounter when instrumenting enterprise DRE systems.

3.1. Overview of OASIS

The design challenges in Section 2 focus on the ability to handle metrics (*e.g.*, collect, extract, and analyze) without knowledge of the metrics of interest (*e.g.*, their structure and quantity to the underlying middleware and system infrastructure). To address these challenges, the OASIS service-oriented middleware can instrument enterprise DRE systems to collect and extract metrics of interest without knowledge of their structure or quantity. Metric collection and extraction in OASIS is also independent of specific technologies and programming languages, which decouples OASIS from enterprise DRE system software implementation details. DRE system developers and testers are thus not constrained to make decisions regarding what metrics to collect for performance analysis tools during the design phase of the system.

Figure 2 presents an high-level overview of OASIS. As

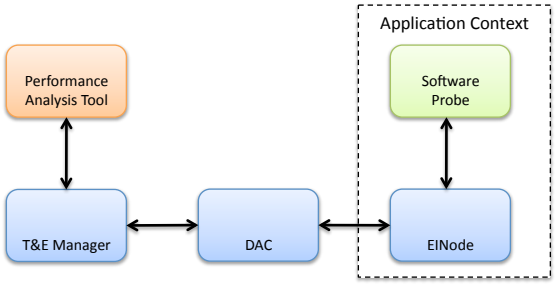

Figure 2. Overview of OASIS

shown in this figure, OASIS consists of the following five entities:

Software probe is an autonomous agent [6] that collects metrics of interest (such as the current value(s) of an event, the current state of a component, or the heartbeat of a component or the node hosting the component) and acts independent of the application being monitored and other entities in OASIS. For example, a heartbeat software probe in the SPRING scenario has an active object that periodic sends metrics that represent a heartbeat, whereas an event monitor probe may send metrics each time a component receives/sends an event.

OASIS supports application-level and system-level probes. Application-level probes are embedded into an application to collect metrics, such as the state of an application component or number of events it sends/receives. System-level probes collect metrics that are not easily available at the application-level or may collect redundant metrics at the application-level, such as current memory usage or heartbeat of each host in the target environment. Both application- and system-level probes submit their metrics to the embedded instrumentation node (EINode) described below. Finally, each software probe is identifiable from other software probes by a user-defined UUID and corresponding human-readable name.

Embedded instrumentation node (EINode) is responsible for receiving metrics from software probes. OASIS has one EINode per application-context, which is a domain of commonly related data. Examples of an application-context include a single component, an executable, or a single host in the target environment. The application-context for an EINode, however, is locality-constrained to ensure data transmission from a software probe to an EINode need not cross network boundaries, only process boundaries, for efficiency. Moreover, the EINode controls the flow of data it receives from software probes and submits to the data and ac-

quisition controller described next. Finally, each EINode is uniquely identifiable from other EINodes by an user-defined UUID and corresponding human-readable name.

Data acquisition and controller (DAC) receives data from an EINode and archives it for acquisition by performance analysis tools, such as querying the latest state of component in the SPRING scenario. The DAC is a persistent database with a static location in the target environment that can be located via a naming service [7]. This design decouples an EINode from a DAC and enables an EINode to dynamically discover at creation time which DAC it will submit data. Moreover, if a DAC fails during at runtime the EINode can (re)discover a new DAC to submit data. Finally, the DAC registers itself with the test and evaluation manager (described next) when it is created and is identifiable by an user-defined UUID and corresponding human-readable name.

Test and evaluation manager (T&E) is the main entry point for user applications (see below) into OASIS. The T&E manager gathers and correlates data from each DAC that has registered with it. The T&E manager also enables user applications to send signals a software probe to alter its behavior at runtime, *e.g.*, decreasing/increasing the hertz of the heartbeat software probe in the SPRING scenario. .

Performance analysis tools interact with OASIS by requesting metrics collected from different software probes via the T&E manager. They can also send signals/-commands to software probes to alter their behavior at runtime. This design enables DRE system developers and testers and performance analysis tools to control the effects of software instrumentation at runtime and minimize the affects on overall system performance, such as ensuring the SPRING scenario's critical path meets its deadline while under instrumentation.

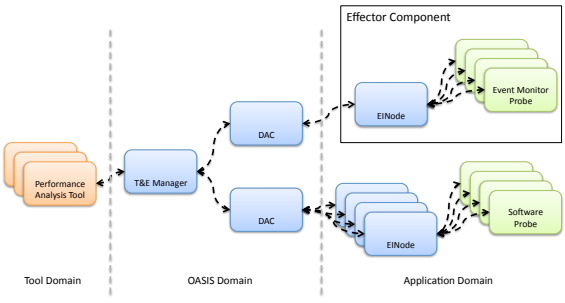

Figure 3. Deployment of OASIS in the SPRING Scenario

Figure 3 shows an example deployment of the OASIS entities above in the context of an Effector component from the SPRING scenario. This figure also shows how DRE system developers and testers write domain-specific software probes—at either the application or system level—that collect metrics unknown to the OASIS middleware, such as probe that monitors incoming events or the heartbeat of a component. While the system under instrumentation is executing in its target environment, software probes collect metrics and submit them to an EINode. The EINode, in turn, submits the data to the DAC, which stores the metrics until user applications request metrics collected from different software probes via the T&E manager.

Figure 3 also highlights the application and network boundaries of OASIS. Software probes do not submit data across different network boundaries; this is the responsibility of the EINode to localize instrumentation overhead on the host. Likewise, the DAC and T&E manager are designed to execute on hosts separate from those running the system under instrumentation to ensure data management concerns of the DAC and T&E manager do not interfere with the system's performance.

The remainder of this section discusses how OASIS addresses the design challenges of enabling dynamic instrumentation without design-time knowledge of the collected metrics or being bounded to a specific technologies and programming language.

3.2. Addressing the Dynamic Instrumentation Challenges in OASIS

3.2.1. Solution 1. Dynamic collection of metrics. As discussed in Section 3.1, OASIS has no knowledge of what metrics are collected during system instrumentation. Instead, DRE system developers and testers implement software probes and register them with an EINode that ensures its data is properly collected by the OASIS middleware. Figure 4 shows the UML class diagram for a software probes interface. Each software probe has `init()`

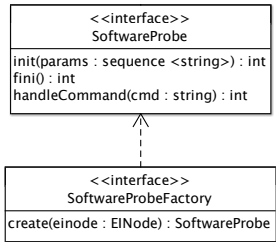

Figure 4. UML Class Diagram for OASIS's Software Probe Architecture

and `fini()` methods to initialize and finalize a software

probe, respectively. A software probe can optionally implement the `handleCommand()` method, which allows it to process commands sent from performance analysis tools via the T&E manager. Finally, each software probe has a corresponding `SoftwareProbeFactory` to support dynamic creation and linking into the application via the Component Configurator pattern [3] and associating the software probe with an EINode to submit metrics.

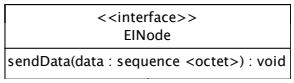

Figure 5. UML Class Diagram for OASIS's EINode

The EINode passed to the `create()` method of the `SoftwareProbeFactory` provides a well-defined interface so any `SoftwareProbe` can submit collected metrics. Figure 5 therefore presents the standard interface for an EINode. As shown in this figure, the EINode exports a single method to the software probe named `sendData()` with a single input parameter of type `BinaryData`, *i.e.*, an array of bytes. When a software probe is ready to submit metrics for collection, it packages the `BinaryData` using the specification presented in Table 1 and invokes the EINode's `sendData()` method.

Name (size)	Description
probeUUID (16)	User-defined UUID
tsSec (4)	Seconds value of timestamp
tsUsec (4)	Micro-seconds value timestamp
seqeunceNum (4)	Metric sequence number
probeState (4)	Probe-defined value (if applicable)
dataSize (4)	Size of metric data

Table 1. Key Elements in Data Packaging Specification for OASIS Software Probes

Table 1 shows how each software probe packages its metrics to include information for OASIS to learn the metric's origin, *i.e.*, the UUID of the software probe, and the probe/metric's state at collection, *e.g.*, collection timestamp, sequence number of metrics, probe state, and metric's data size. The remaining contents of the `BinaryData`, such as the actual metrics, are outside of the EINode's concern and remain unknown.

After an EINode receives packaged metrics from a software probe, the EINode submits it to the DAC. The behavior of an EINode for submitting collected metrics to the DAC can vary between different implementations. Regardless of an EINode's implementation, the specification in Table 2 is used to package data received from a software probe into `BinaryData`. An interface similar to the EINode (see Figure 5) is then used to send the packaged `BinaryData` object to a DAC.

Name (size)	Description
magic (3)	Identification of OASIS packet
byteOrder (1)	Byte-order of data
versionNumber (2)	OASIS version number
reserved (2)	padding for word alignment
einodeUUID (16)	EInode unique id

Table 2. Key Elements in Data Packaging Specification for OASIS EINode

Table 2 shows how the EINode prepends to `BinaryData` the header information OASIS needs to learn its application-context origin. The EINode then submits the data to the DAC for storage. If metrics sent to a DAC are packaged according to Table 1 and Table 2 OASIS can dynamically collect metrics without knowing their details. This approach also decouples OASIS from any specific technology or programming language since data are a well-defined binary stream produceable using any language or platform that supports sockets—thereby addressing Challenge 1 in Section 2.

3.2.2. Solution 2. Discovery of metrics for analysis. In many cases, performance analysis tools requesting data from OASIS via the T&E manager will know at design time the metrics being collected by software probes submitting data to an EINode. In other cases, such tools may want to request data for metrics not known at design-time. For example, in the SPRING scenario, DRE system developers and testers want to construct a generic GUI to monitor all metrics collected while the system is instrumented in its target environment. Since the GUI does not know all the different metrics collected by software probes at design time, the GUI needs mechanisms to learn metrics at runtime.

We address the challenge of discovering metrics for analysis without design-time knowledge by requiring each OASIS entity shown in Figure 2 to perform a registration and unregistration process at startup and shutdown time, respectively. The registration process between an EINode and a DAC, as well as a DAC with a T&E manager, involves understanding the composition of metric collection (*i.e.*, the origin and path of metrics collected during system instrumentation). The registration process for a software probe with an EINode, however, is more critical because this is when the software probe notifies OASIS metric's format.

```
<?xml version='1.0' ?>
<xsd:schema>
  <xsd:element name='probeMetadata'
               type='component.state' />
  <xsd:complexType name='component.state'>
    <xsd:annotation id='metadata'>
      <xsd:appinfo>
        0A499B6B-7250-4B88-B9DC-360D32639081
      </xsd:appinfo>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name='component'
                   type='xsd:string' />
      <xsd:element name='state'
                   type='xsd:integer' />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Listing 1: Example Registration File for Software Probe in OASIS

Listing 1 presents an example registration for a software probe from the SPRING scenario that keeps track of a component's state (*e.g.*, activated or passivated). Each software probe's registration in this listing is a XML schema definition (XSD) file. We use XSD since it provides a detailed description of data types, such as quantity and constraints, which can help with optimizations and filtering/managing data. The element with the name `probeMetadata` defines the root element that describes the format of the metric collected by a software probe. Likewise, the child annotation of the `probeMetadata` element with the id named `metadata` describes information about the software probe, such as the user-defined UUID and description of the software probe.

During registration, a software probe passes its XSD file to the EINode when the `create()` method is invoked on its `SoftwareProbeFactory` (see Listing 4). The EINode then passes the XSD file to the DAC with which it is registered. Performance analysis tools can request registration information for a software probe via the T&E manager, which includes the metadata describing its metric format. Using the metadata for a software probe, user applications can learn about metrics at runtime for analytical purposes—thereby addressing Challenge 2 in Section 2.

4. Evaluating OASIS's Runtime Flexibility and Instrumentation Overhead

This section analyzes the results of experiments that empirically evaluate the capabilities and performance of OASIS in the context of the SPRING scenario presented in Section 2, which is a representative enterprise DRE system composed of programming languages and technologies that executes in heterogeneous environments where computers can run many different operating systems. The heterogeneity of the SPRING scenario influenced the design of OASIS, as discussed in Section 3.

4.1. Experiment Setup

A key concern of developers of applications and middleware for DRE systems is that dynamic instrumentation will negatively impact existing QoS properties, such as response-time and utilization, of the instrumented system. Due to the design of OASIS (described in Section 3), QoS properties related to the DAC and the T&E Manager do not impact existing QoS properties of the system undergoing instruction, such as the application in the SPRING scenario. Instead, the EINode and software probes have more impact on QoS properties for instrumented system because the DAC and T&E manager are not deployed in the application domain. In contrast, the EINode and software probes interact directly with the instrumented system.

Since the EINode and software probes have more of an impact on existing QoS properties for the system undergoing instrumentation, developers and testers of the SPRING scenario were interested in determining if they could use OASIS to monitor the application in real-time without causing the application in the SPRING scenario to miss its critical path deadline. In particular, system testers want to monitor (1) as many events sent between each component and (2) the heartbeat of each application. Monitoring these two metrics greatly impacts the end-to-end response time of the application in the SPRING scenario and evaluate OASIS's capabilities.

System testers therefore used the CUTS system execution modeling tool [5] to construct the application in the SPRING scenario shown in Figure 1. CUTS was used to (1) model the behavior and workload of each component at a high-level of abstraction and (2) auto-generate source code for the CIAO architecture from constructed models. System testers then compiled the generated source code on its target architecture and emulated the system on a cluster of computers running in ISISlab (`www.isislab.vanderbilt.edu`). Each computer in ISISlab used Emulab software to configure the Fedora Core 8 operating system. Likewise, each computer is an IBM Blade Type L20, dual-CPU 2.8 GHz processor with 1 GB RAM.

4.2. Experiment Results

Section 4.1 provided background information that the system testers of the SPRING scenario were interested in executing. In particular, the system testers wanted to understand how using the dynamic instrumentation capabilities of OASIS would affect end-to-end response time of the application in the SPRING scenario. Figure 6 shows a single test run of the application in the SPRING scenario in ISISlab, where system testers adjusted the hertz of the heartbeat software probe. As shown in the figure, system testers initially started with a configuration of 1 hertz (or 1 event/sec)

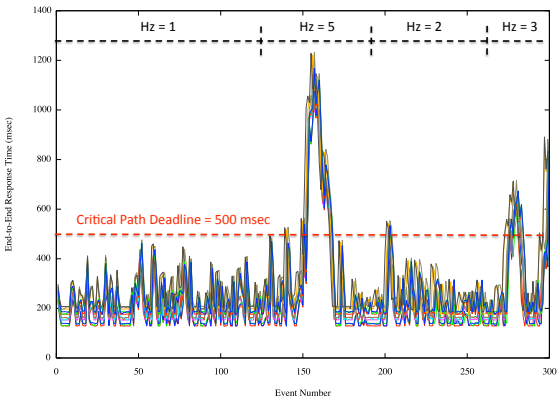

Figure 6. Single Test Run of Adjusting the Heartbeat Software Probe's Hertz in SPRING Scenario's Application Under OASIS instrumentation.

for each heartbeat software probe embedded in a component. Since the end-to-end response time for the application's critical path of execution between the Effectors and Sensors was under its 500 msec deadline for this particular scenario, the system testers decided to increase the heartbeat software probe's hertz to improve awareness of application liveliness. After increasing the heartbeat to 5 hertz, the end-to-end response time began increasing above 500 msec, as shown in Figure 6.

Since the heartbeat probe negatively impacted end-to-end response time for the application's critical path of execution, system testers decreased the heartbeat software probe's hertz to 2. Figure 6 shows how this dynamic change enabled them to increase awareness of application liveliness and not negatively impact end-to-end response of the application's critical path of execution. System testers also increased the heartbeat software probe's hertz to 3 to determine the new configuration would impact end-to-end response time. Unfortunately, 3 hertz caused the end-to-end response time to gradually increase with respect to time. System testers therefore learned that they could configure the heartbeat software probe to execute at 2 hertz without impacting end-to-end response time of the application's critical path of execution.

The results above are just one of many different executions of the SPRING scenario. Experience gained from running this scenario throughout the system lifecycle underscores the challenges of validating OASIS's usage in enterprise DRE systems. In general, OASIS's dynamic capabilities can adapt to different application domains and its generic instrumentation and data collection facilities help remove complexities associated with distributed instrumentation and data collection in enterprise DRE systems. To truly leverage OASIS's capabilities, however, DRE system testers will need to discover patterns for instrumenting distributed systems, similar to patterns of software architectures [1], and techniques for optimizing instrumentation and data collection concerns in such systems.

5. Related Work

This section compares and contrasts our work on OASIS with related research on techniques for instrumentation and generic metric collection.

Instrumentation techniques. Tan et al. [13] discuss methodologies to verify instrumentation techniques for resource management. Their approach decomposes instrumentation for resource management into monitor and corrector components. OASIS also has a monitor component (*i.e.*, software probes) and a corrector component (*i.e.*, user applications). OASIS extends their definition of instrumentation components, however, to include those necessary to extract metrics from the system in an efficient and effective manner (*i.e.*, the EINode, DAC, and T&E manager).

DTrace [2] is a non-intrusive dynamic instrumentation utility for production systems that has similar concepts as OASIS (such as application- and system-level software probes). DTrace, however, is locality constrained, whereas OASIS can collect data in a distributed environment. DTrace and OASIS can both collect metrics without design time knowledge of what metrics are being collected via instrumentation. DTrace also has the option of filtering instrumentation data at the software probe level using predicates. OASIS can achieve similar functionality—and greater flexibility—at the software probe, DAC, and T&E manager levels if each component uses a software probe's registered metadata to learn and filter collected metrics at runtime. Finally, it is believed that DTrace and OASIS can complement each other to remove the locality constrained characteristics of DTrace.

Generic metric collection techniques. XML is the basis of several techniques for collecting metrics without *a priori* knowledge of their type. XML is used in technologies such as Simple Object Access Protocol (SOAP) for Web Services, XML-RPC, and XML Metadata Interchange (XMI). OASIS likewise enables collection/transmission of metrics without design time knowledge of their type. OASIS uses raw binary streams to collect and transmit metrics, however, as opposed to verbose text strings as in XML that can impact the performance of an enterprise DRE system. OASIS uses XML to describe metametrics, which are metadata that describes the metrics structure, data types, and quantity, collected and transmitted at registration time (*i.e.*, before the system is fully operational).

Generic data types, such as CORBA Any and Java Object, can be used to dynamically collect and transmit metrics. OASIS improves upon these techniques since it is not bound to a particular programming language or middleware technology. Moreover, using generic types to collect and transmit metrics makes it hard for receivers (such as user applications) to learn about unknown types unless the language supports built-in discovery mechanisms, such as reflection. OASIS removes this complexity by storing metadata about a software probe's metrics (which is kept separate from the actual metrics) so programming languages and technologies used to implement user applications in OASIS that do not support type discovery mechanisms can still learn about metric types at runtime.

6. Concluding Remarks

Conventional techniques for instrumenting enterprise DRE systems and determining which metrics to collect for performance analysis tools can limit their analytical capabilities. Moreover, deferring design decisions related to instrumenting DRE systems and determining what metrics to collect can limit the metrics available to performance analysis tools since the instrumentation may not be incorporated into the system's design. To address these limitations, this paper presented OASIS, which is service-oriented dynamic instrumentation middleware that enables enterprise DRE system developers and testers to collect metrics via instrumentation without prematurely committing to tightly-coupled design decision during the early phases of a system's lifecycle. The results of our experiments show how OASIS helps adapt the instrumentation needs of enterprise DRE system developers and testers by enabling them to control the impact of instrumentation overhead at runtime.

The following are our lessons learned thus far based on our experience of applying OASIS to the SPRING scenario:

1. Dynamic configuration of probes at runtime minimizes probe effects. OASIS's ability to alter the behavior of software probes at runtime helps minimize instrumentation overhead, *e.g.*, software probes have essentially the same effects on QoS if they are present or not. Enterprise DRE system developers and testers therefore have greater levels of confidence they can incorporate instrumentation into production systems and still meet the system's QoS requirements. Our future work will investigate techniques to automate minimizing probe effects for dynamic instrumentation of enterprise DRE systems.

2. Separating metadata from data improves discovery capabilities. Since OASIS separates metadata (*i.e.*, the XSD files) from the actual data (*i.e.*, collected metrics) for each software probe, performance analysis tools to utilize collected metrics at runtime without having prior knowledge of its structure and quality. Our future work will investigate techniques to optimize OASIS's data collection, archiving, and retrieval capabilities by leveraging this metadata.

OASIS is integrated into the CUTS system execution modeling tool, which are available for download in open-source form from `www.cs.iupui.edu/CUTS`.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[2] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the General Track: 2004 USENIX Annual Technical Conference*, pages 15–28, June 2004.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[4] J. H. Hill, T. Silveria, J. M. Slaby, , and D. C. Schmidt. The SPRING Scenario: An Heterogeneous Enterprise Distributed Real-time and Embedded (DRE) System Case Study. Technical Report TR-CIS-1207-09, Indiana University-Purdue University Indianapolis, December 2009.

[5] J. H. Hill, J. Slaby, S. Baker, and D. C. Schmidt. Applying System Execution Modeling Tools to Evaluate Enterprise Distributed Real-time and Embedded System QoS. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing Systems and Applications*, Sydney, Australia, August 2006.

[6] H. S. Nwana. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3):1–40, 1996.

[7] Object Management Group. *Naming Service, version 1.3*, OMG Document formal/2004-10-03 edition, October 2004.

[8] K. O'Hair. The JVMPI Transition to JVMTI. `java.sun.com/developer/technicalArticles/Programming/jvmpitransition`, 2006.

[9] T. Parsons. *Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems*. PhD thesis, University College Dublin, Belfield, Dublin 4, Ireland, 2007.

[10] M. Pezzini and Y. V. Natis. Trends in Platform Middleware: Disruption Is in Sight. `www.gartner.com/DisplayDocument?doc_cd=152076`, September 2007.

[11] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[12] A. Tamches and B. P. Miller. Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. *International Journale High Performance Computing Applications*, 13(3):263–276, 1999.

[13] Z. Tan, W. Leal, and L. Welch. Verification of Instrumentation Techniques for Resource Management of Real-time Systems. *J. Syst. Softw.*, 80(7):1015–1022, 2007.