OASIS: An Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems

James Hill*, Hunt Sutherland†, Paul Staudinger†, Thomas Silveria‡, Douglas C. Schmidt§, John Slaby‡ and Nikita Visnevski†
* *Indiana University-Purdue University Indianapolis*
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

† *GE Global Research*
Niskayuna, NY USA
Email: {sutherland,staudinger}@crd.ge.com
visnevsk@research.ge.com

‡ *Raytheon Company*
Portsmouth, RI USA
Email: {thomas_silveria,john_m_slaby}
@raytheon.com

§ *Vanderbilt University*
Nashville, TN USA
Email: d.schmidt@vanderbilt.edu

***Abstract*—Instrumentation is a critical part of evaluating an enterprise distributed real-time and embedded (DRE) system's performance. Traditional techniques for instrumenting enterprise DRE systems require DRE system developers to make design decisions regarding what metrics to collect during early phases of the software lifecycle so these needs can be factored into the system architecture. In many circumstances, however, it is hard for DRE system developers to know this information during early phases of the software lifecycle—especially when metrics come from many heterogeneous sources (such as application- and system-level hardware and software resources) and evaluating performance is traditionally an after-thought.**

To address these issues, this article presents the design and performance of OASIS, which is SOA-based middleware and tools that dynamically instruments enterprise DRE system without requiring design-time knowledge of which metrics to collect. This article also empirically evaluates OASIS in the context of a representative enterprise DRE system case study from the domain of shipboard computing. Results from applying OASIS to this case study show that its flexibility enables DRE system testers to precisely control instrumentation overhead. We also highlight open challenges in dynamic instrumentation for next-generation enterprise DRE systems.

***Keywords*-dynamic instrumentation, enterprise DRE systems, service-oriented architecture, middleware, real-time instrumentation**

I. Introduction

Enterprise distributed real-time and embedded (DRE) systems (*e.g.* 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.*, component-base middleware and service-oriented middleware [26]) and deployed into heterogeneous execution environments. In addition to meeting their functional requirements, enterprise DRE systems must also meet non-functional requirements, such as latency, response time, and scalability, so that applications achieve their expected behavior in a timely manner [32]. Failure to meet non-functional requirements can cause the system to not meet its quality-of-service (QoS) guarantees.

To ensure enterprise DRE system QoS, DRE system stakeholders, such as testers, developers, and end-users, monitor the system's behavior and resources (*e.g.*, CPU usage, memory allocation, and event arrival rate) and analyze collected metrics. Conventional approaches for monitoring enterprise DRE system behavior and resources employ software instrumentation techniques [4], [24], [29], [32] 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) then evaluate collected metrics and inform DRE system stakeholders if the system is meeting its QoS expectations, such as honoring deadlines, processing events in a timely manner, not hogging to much memory. These tools can also identify QoS bottlenecks and application components that exhibit high and/or unpredictable behavior and resource usage [25].

Although software instrumentation helps ensure QoS guarantees, conventional techniques [32] for collecting metrics are often tightly coupled to system implementations. As a result, DRE system developers must decide what metrics to collect during the system design phase. These concerns are also factored into the verification and validation process for system implementations [32]. DRE system developers and testers therefore are faced with the following drawbacks as a result of using a tightly-coupled approach to software instrumentation:

- Developers often do not know *a priori* what metrics are needed to evaluate system QoS—especially since performance is an *afterthought* [12] (*i.e.*, evaluated late in the software lifecycle);
- Incorporating new metrics late in the software lifecycle

The author would like to thank the Test Resource Management Center (TRMC) Test and Evaluation/Science and Technology (T&E/S&T) Program for their support. This work is funded by the T&E/S&T Program through the Naval Undersea Warfare Center, Newport, RI, contract N66604-05-C-2335.

typically triggers verification and validation of the changes system, which is costly [34];
- Developers continually reinvent instrumentation logic across different application domains; and
- Developers apply *ad hoc* techniques, such as augmenting existing code to inject the instrumentation hooks, without understanding the impact on overall system design and maintainability.

To overcome the drawbacks with conventional enterprise DRE system instrumentation techniques, there is need for improved techniques that do not require DRE system stakeholders to make software instrumentation design decisions during early phases of software lifecycle. A promising approach to realizing the goal is *dynamic instrumentation* [4], [31], where developers control what and how metrics are collected from the system at runtime rather than during the design phase. When dynamic instrumentation methodologies are combined with service-oriented middleware methodologies, many challenges associated with dynamic instrumentation, such as data collection and extraction, can be decoupled from—and resolved independently of—the core application business logic. Moreover, these solutions can be reused across different application domains.

This article presents the *Open-source Architecture for Software Instrumentation of Systems* (OASIS) and provides the following contributions to R&D on dynamic instrumentation of enterprise DRE systems:

- It describes the design challenges and OASIS-based solutions associated with realizing dynamic instrumentation middleware that is suitable for enterprise DRE systems;
- It shows how OASIS has been applied to a representative enterprise DRE system case study from the domain of shipboard computing environments; and
- It analyzes empirical results that quantify how OASIS enables DRE system developers to dynamically configure the instrumentation middleware *manually* to minimize impact on end-to-end response time.

This article extends the seminal paper on OASIS [9] by:

- Presenting a more detailed overview of OASIS's data collection architecture;
- Discussing how the generalized architecture is customizable for different application domains, programming languages, and distributed technologies; and
- Presenting open challenges for future research on dynamic instrumentation of enterprise DRE systems.

Article organization. The remainder of this article is organized as follows: Section II motivates OASIS via a case study from the domain of shipboard computing; Section III describes the structure and functionality of OASIS; Section IV presents empirical results that evaluate OASIS's impact on QoS requirements from the case study; Section V compares work on OASIS with related efforts; and Section VI presents concluding remarks and lessons learned.

II. Case Study: The SPRING Scenario

This section motivates the need for OASIS via a case study based on the SPRING scenario. This scenario is a representative enterprise DRE system from the domain of shipboard computing composed of programming languages and technologies that execute in heterogeneous environments where computers can run different operating systems, as shown in Figure 1.

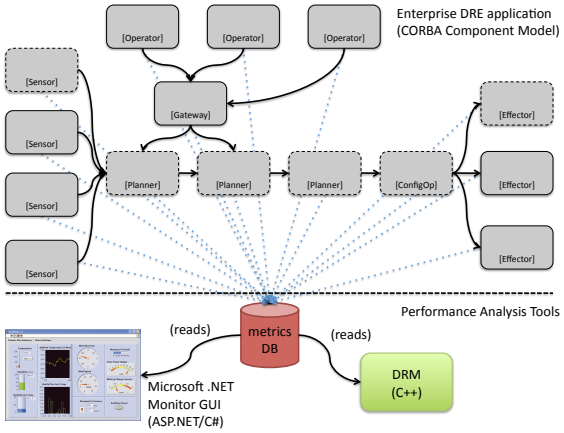

Figure 1. Overview of the SPRING Scenario

This scenario tests QoS concerns (such as throughput, 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 [7].

Existing techniques for instrumenting enterprise DRE systems like 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, impedes flexibility in system design and implementation throughout the software lifecycle, *e.g.*, by complicating changes to the number/type of components and precluding the refreshing of new technologies due to the effort needed to revalidate QoS requirements.

In general, developers and testers of the SPRING scenario faced the following challenges:

• **Challenge 1: Dynamic collection and extraction of metrics without design time knowledge metric composition.** While the SPRING scenario is being developed it is hard for DRE system developers and testers in the early

phases of the software lifecycle to anticipate all the metrics they will need to collect for performance analysis. 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 (1) collect and extract metrics without making such decisions at design time and (2) dynamically modify collection to ensure instrumentation does not unduly perturb performance. Section III-B1 shows how OASIS addresses this challenge by using a multi-level packaging technique that enables 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 J2EE [30], CORBA [21]–[23], DDS [19], and Microsoft .NET [13], 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 like 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 III-B2 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.

• **Challenge 3: Real-time reporting of collected metrics.** The ability to dynamically instrument and collect metrics without design time knowledge of what metrics are being collected is not complete without the ability to analyze such metrics in real-time. For example, performance analysis tools may need to monitor system performance in real-time and adapt the system's behavior as needed, such as changing the rate of collecting information without impacting system response time or migrating software components to reduce CPU workload or improve fault tolerance [33]. DRE system developers therefore need architectural support that enables them to receive collected information in real-time for actuator purposes. Section III-B3 shows how OASIS addresses this challenge using publisher-subscriber services that allow performance analysis tools to register for a software probe and receive metrics in real-time (*i.e.*, as it is collected).

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.

III. The Structure and Functionality of OASIS

This section describes the structure and functionality of OASIS, focusing on how it addresses the key challenges presented in Section II that DRE system developers and testers encounter when dynamically instrumenting enterprise DRE systems.

A. Overview of OASIS

The challenges in Section II focus on the ability to handle(*e.g.*, collect, extract, and analyze) metrics without *a priori* knowledge of metric details (*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.

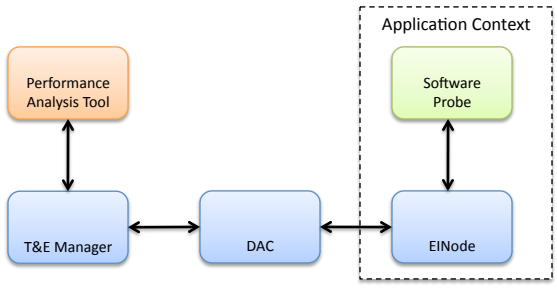

Figure 2. Overview of OASIS

Figure 2 presents an high-level overview of OASIS. As shown in this figure, OASIS consists of the following five entities:

Software probe is an autonomous agent [17] 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 monitored application and other entities in OASIS. For example, a heartbeat software probe in the SPRING scenario has an active object that periodically sends metrics representing 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 (such

as current memory usage or the heartbeat of each host in the target environment) that are not easily available at the application-level or may be redundant the application-level. Both application- and system-level probes submit their metrics to the *Embedded Instrumentation Node (EINode)* described below. 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. Application-context examples include a single component, an executable, or a single host in the target environment. The application-context for an EINode, however, is locality-constrained for efficiency, *i.e.*, to ensure data transmission from a software probe to an EINode need only cross process boundaries—rather than network boundaries. Moreover, the EINode controls the flow of data it receives from software probes and submits to the data and acquisition controller described next. Each EINode is uniquely identifiable from other EINodes by a user-defined UUID and corresponding human-readable name.

Data Acquisition and Controller (DAC) receives data from an EINode and archives it for subsequent 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 [20]. This design decouples an EINode from a DAC and enables an EINode to discover which DAC it will submit data to dynamically at creation time. Moreover, if a DAC fails during at runtime the EINode can (re)discover a new DAC to submit data. 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 shows an example deployment of the OASIS

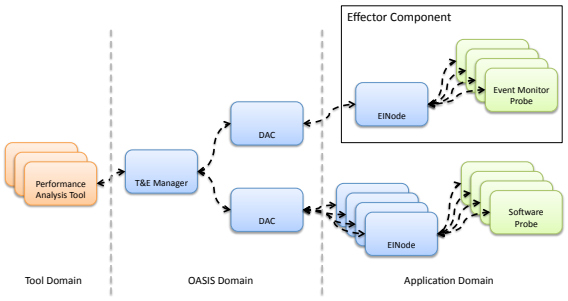

Figure 3. Deployment of OASIS in the SPRING Scenario

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 directly; instead, the EINode localizes instrumentation overhead on the host. Likewise, the DAC and T&E Manager are designed to execute on hosts separate from those running the instrumented system 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 challenges of enabling dynamic instrumentation without design-time knowledge of the collected metrics or being bounded to a specific technologies and programming language.

B. Addressing the Dynamic Instrumentation Challenges in OASIS

1) Solution 1. Dynamic Collection of Metrics: As discussed in Section III-A, 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 collected properly by the OASIS middleware. Figure 4 therefore shows OASIS's architecture for facilitating dynamic collection metrics from software probes. As shown in this figure, the architecture is composed of several interfaces. Each interface, except for the `SoftwareProbe` interface, represents a point where different implementations can be inserted into the architecture to provide more domain-

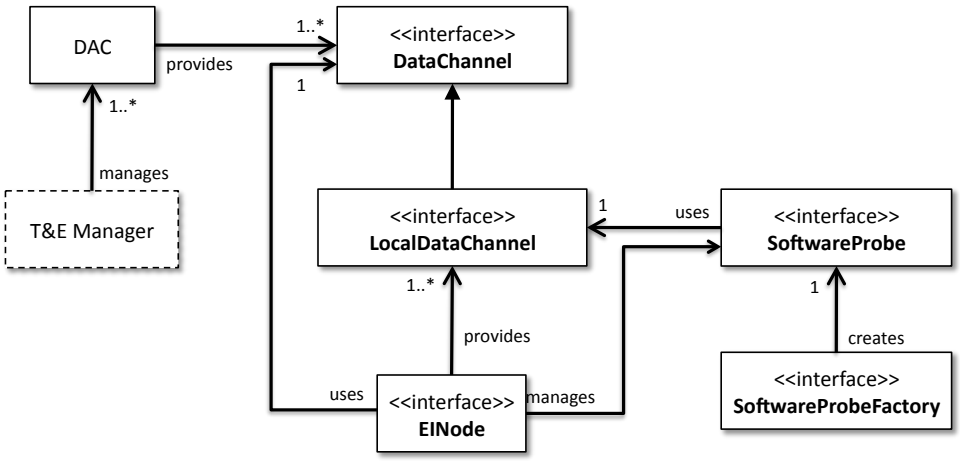

Figure 4. OASIS's Architecture Diagram for Dynamic Collection of Metrics.

specific behavior, *e.g.*, data channels that intelligently select metrics to transmit when dealing with software probes that have high data rates [5].

Figure 5 shows the UML class diagram for the `SoftwareProbe` interface. As shown in this figure, each software probe implements an `init()` and `fini()` method 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. For example, if a software probe automatically sends updates to its host EINode, then a performance analysis tool can send a command to change its update rate, *i.e.*, the software probes hertz. Such commands can also be used to change the software probe's configuration based on the state of the software system undergoing instrumentation.

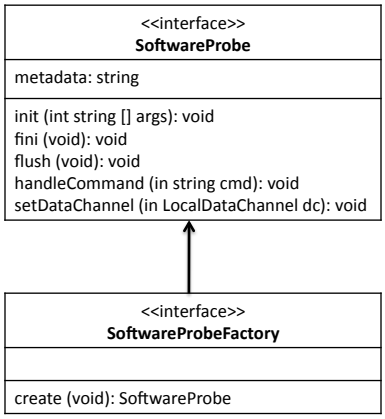

Figure 5. UML Class Diagram for OASIS's Software Probe Architecture

Each software probe implements a `flush()` method that allows the host EINode or application to force the software probe to send its most recently collected metrics to its hosting EINode. The `setDataChannel()` method determines the local data channel that the software probe uses to send collected metrics to the EINode. This method therefore is invoked by the hosting EINode when a software probe is loaded into memory, but before the software probe's `init()` method in invoked.

`LocalDataChannel` (see Figure 4 and Figure 7) is an interface for a locality-constrained object. Its implementation can therefore vary, depending on the needs of the enterprise DRE system undergoing dynamic instrumentation. For example, the local data channel implementation can aggregate collected metrics before sending them to the DAC to reduce network overhead. Each software probe has a corresponding `SoftwareProbeFactory` to support dynamic creation and linking into the application via the Component Configurator pattern [28] and associating the software probe with an EINode—via an implementation of the `LocalDataChannel` interface—to submit metrics.

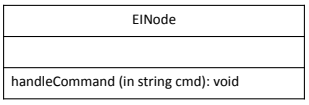

Figure 6. UML Class Diagram for OASIS's EINode

Figure 6 presents the standard and lightweight object structure for an EINode. As shown in this figure, the EINode exports a single method to the software probe named `handleCommand()`, which receives commands from a DAC. The commands received via this method have the format: `[PROBE] [COMMAND]`, where `PROBE` is the name of the target software probe that receives the command and `COMMAND` is the command-line passes to the target probe, if found.

The EINode also implements the one or more

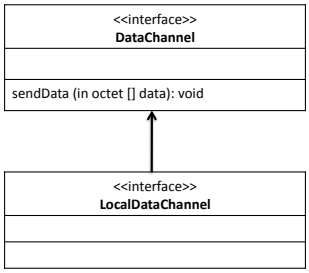

Figure 7. UML Class Diagram for OASIS's Data Channels

`LocalDataChannel` objects that are used by different software implementation probes. As shown in Figure 7, the `LocalDataChannel` has a single method named `sendData()` used by the software probe to send packaged data in binary format based on the specification presented in Table I to the EINode. As shown in this table, 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.

Table I
KEY ELEMENTS IN DATA PACKAGING SPECIFICATION FOR OASIS SOFTWARE PROBES

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

After an EINode receives packaged metrics from a software probe, the EINode submits it to the DAC via an exposed `DataChannel` interface (see Figure 7). This version of the data channel interface is designed to support network communication, unlike the `LocalDataChannel` since the DAC and EINode reside on different machines (as shown in Figure 3). 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 II is used to package data received from a software probe into binary data. The EINode then invokes the `sendData()` method on the `DataChannel`, which sends collected metrics to the target DAC.

Table II shows the necessary header information the EINode prepends to binary data it receives from a software probe. This information allows OASIS to learn the application-context origin for collected metrics. The EINode then submits the data to the DAC for storage. OASIS can

Table II
KEY ELEMENTS IN DATA PACKAGING SPECIFICATION FOR OASIS EINODE

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

dynamically collect metrics without knowing their details as long as metrics sent to a DAC are packaged according to Table I and Table II. 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 II.

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, these 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 is more critical, however, because this is when the software probe notifies OASIS metric's format.

```
1  <?xml version="1.0"?>
2  <xsd:schema>
3    <xsd:complexType name="Component.State">
4      <xsd:sequence>
5        <xsd:element name="name" type="xsd:string" />
6        <xsd:element name="state" type="xsd:byte" />
7      </xsd:sequence>
8    </xsd:complexType>
9
10   <xsd:element
11       name="probeMetadata" type="Component.State" />
12 </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) that can help with optimizations and filtering/managing data. The element named `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 5). 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 II.

3) Solution 3. Pub/sub for Real-time Reporting: In many cases, instrumentation of a software system is done to collect metrics for postmortem analysis, such as debugging the application or locating data trends in performance properties. Moreover, dynamic software instrumentation provides the ability to control at run-time the instrumentation's behavior, which implies that performance analysis tools should have the ability to control this behavior. Such control, however, is typically based on analyzing collected metrics at real-time and invoking commands back into the system—similar to an actuator.

OASIS address this challenge problem by implementing the Publisher-Subscriber [3] pattern that allows performance analysis tools to register for metrics collected by software probes. When software probes send new data to the DAC, if the software probe has subscribers, the metrics are pushed to the subscriber. OASIS implements the Publisher-Subscriber pattern using CORBA, unlike the rest of OASIS that is not tied to a platform or architecture. This design choice was made in the specification and architecture for the following reasons:

- The technology-, architecture-, and platform-independence is needed only for extracting metrics from the enterprise DRE system. Once metrics reach the DAC, it is outside the application-space of the system undergoing instrumentation.
- The performance analysis tool and the DAC typically will reside on different host machines, which implies that there will be network communication. CORBA therefore simplifies inter-network communication concerns involving objects on residing on different machines.

Listing 2 shows the publisher-subscriber interface implemented by the T&E Manager and DAC, respectively. As shown in this listing the T&E Manager must implement `register()` and `unregister()` methods. The registration method requires an target DAC and software probe. Likewise, the performance analysis tool must implement a `DataChannel` interface—similar to the `DataChannel` implemented by the DAC, which allows the performance analysis tool to provide a domain-specific implementation of the data channel, if necessary.

```
1  // Publisher interface for the T&E Manager.
2  interface TnEPublisher {
3    Cookie register (in UUID dac,
4                     in UUID probe,
5                     in DataChannel subscriber);
6
7    void unregister (in Cookie c);
8  };
9
10 // Publisher interface for the DAC.
11 interface DACPublisher {
12   Cookie register (in UUID probe,
13                    in DataChannel subscriber);
14
15   void unregister (in Cookie c);
16 };
```

Listing 2. Publisher-subscriber interface definition for the DAC and T&E Manager.

The DAC also implements `register()` and `unregister()` methods. The DAC's registration method, however, requires only the target software probe and the provided `DataChannel` interface. The result of successful registration is a cookie, which is a subscription id for each client. This cookie is used to unregister the subscribers.

After successful registration, performance analysis tools begin receiving updates via the provided `DataChannel` interface. Based on collected data, the performance analysis tool can send command to software probes at real-time via the `handleCommand()` method. Performance analysis tools thus have the ability to adapt software instrumentation autonomously based on the needs of the application at runtime—thereby addressing Challenge 3 in Section II.

C. Using a Definition Language to Define Software Probes

Implementing a software probe for OASIS requires the following two steps:

1) Creating the XML Schema Definition that defines the software probes collected metrics, *i.e.*, the software probes metadata.
2) Implementing the software probe to package data according to its metadata.

Software probes can be implemented in different programming languages, such as Java and C++, so it can be used for different application domains.

To simplify implementing software probes, OASIS provides the *Probe Definition Language (PDL)*, which is a domain-specific language for defining metric collected by a software probe. The language specification resembles the CORBA *Interface Definition Language (IDL)*, but uses different keywords to define data. Listing 3 shows the PDL for the component state probe in Listing 1.

```
1  module Component {
2    struct UnusedType {
3      int32 value1;
4      int32 value2;
5    };
6
7    [uuid(0A499B6B-7250-4B88-B9DC-360D32639081);
8     version(1.0)]
9    probe State {
10     string name;
11     byte state;
12   };
13 }
```

Listing 3. Specification of the component state probe using OASIS's PDL.

As shown in this listing, PDL supports tuple definitions as structures. A software probe is also defined by the keyword `probe`. In addition, each probe has a set of attributes that define its implementation properties, such as UUID and version number. After a software probe is defined using PDL, it is compiled into a separate XML Schema Definition and software probe implementation. The current implementation of OASIS generates software probes in C++, but can be extended easily to support other programming languages.

Table III
DIFFERENT DATA TYPES SUPPORTED BY OASIS'S PDL.

Category	Keyword
Integer	int8, int16, int32, int64
Unsigned Integer	uint8, uint16, uint32, uint64
Decimal	real32, real64
String	char, string

Table III list the different data types supported in PDL. Each data type can be specified as an array by using standard bracket notation: `int8 v[N]` where `N` is the number of elements in the array. Likewise, ranges can be specified in the bracket notation: `int8 v[min, max]` where min is the minimum number of elements in the array and max is the maximum number of elements in the array. If $max = infinite$, then there are unbounded number of elements in the array, whereas if $min = 0$ and $max = 1$ the element is optional—similar to XML Schema Definition.

IV. 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 II. The heterogeneity of the SPRING scenario influenced the design of OASIS, as discussed in Section III.

A. 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 III), QoS properties related to the *Data Acquisition and Controller (DAC)* and the *Test and Evaluation (T&E) Manager* do not impact existing QoS properties of the system undergoing instruction, such as the application in the SPRING scenario. Instead, the *Embedded Instrumentation Node (EINode)* and software probes have more impact on QoS properties for instrumented system since 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 wanted to monitor (1) as many events sent between each component and (2) the heartbeat of each application. Monitoring these two metrics significantly impacts the end-to-end response time of the application in the SPRING scenario and is thus essential to evaluate OASIS's capabilities.

System testers therefore used the CUTS system execution modeling tool [8] 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 interconnected via six Cisco 3750G-24TS network switches and one Cisco 3750G-48TS network switch. The remainder of this section analyzes the results of the experiment.

B. Experiment Results

Section IV-A 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 8 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) 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

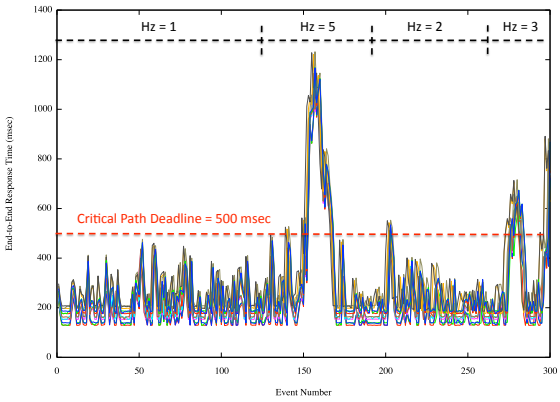

Figure 8. Single Test Run of Adjusting the Heartbeat Software Probe's Hertz in SPRING Scenario's Application Under OASIS Instrumentation.

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 8.

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 8 shows how this dynamic change enabled testers 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.

C. Open Research Problems

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. Based on this experience, the following is a list of open research problems in dynamic software instrumentation of DRE systems.

Patterns for (dynamic) software instrumentation of enterprise DRE systems. Different application domains require different software instrumentation needs, which is part of the reason that software instrumentation concerns have been tightly coupled with the system's implementation. Once this tight-coupling is removed (as done with OASIS) it is possible to separate the core "business-logic" of software instrumentation from the more domain-specific aspects, such as collection, aggregation, and extraction techniques. Moreover, this decoupling will force DRE system developers and testers to discover patterns for instrumenting distributed systems, similar to patterns of software architectures [1]–[3], [10], [28].

Optimization techniques for general-purpose distributed system instrumentation middleware. When a concern, such as software instrumentation, is separated from the overall application it is typically over-generalized and highly configurable. For example, distributed middleware architectures, such as CORBA, Microsoft.NET, and J2EE, removed complexities associated with network communication, through abstraction and generalization. Although these abstractions generally improve reuse and software quality, many times the generalization does not fit every application domain. Similar to general-purpose distributed middleware, DRE system developers and testers must develop different optimization techniques for software instrumentation of enterprise DRE systems that try to minimize resource usage, such as network bandwidth, CPU overhead, memory, while not impacting existing qualities of the system undergoing instrumentation.

Verification and validation of DRE systems that utilize dynamic instrumentation middleware. Verification and validation of dynamic enterprise DRE systems is still an active research area [6], [11], [18], [27]. Dynamic instrumentation middleware adds another level of complexity to the problem because these concerns are not tightly coupled to the design and can change at run-time. These run-time changes, in turn, can impact the integrity of the system if the such changes force the system to comprise its QoS guarantees. DRE system developers and testers therefore need to research methodologies that support V&V of enterprise DRE systems that use dynamic instrumentation middleware capabilities.

V. Related Work

This section compares and contrasts our work on OASIS with related work.

Instrumentation techniques. Tan et al. [32] 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 efficiently and effectively (*i.e.*, the EINode, DAC, and T&E manager).

DTrace [4] 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 is locality constrained, however, 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. DTrace and OASIS can complement each other to remove the locality constrained characteristics of DTrace.

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 decoupled 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 discover and utilize metric types at runtime.

Distributed instrumentation middleware. The Testing and Training Enabled Architecture (TENA) [14]–[16] is a distributed service-oriented middleware that supports instrumentation of distributed software applications—similar to OASIS. The main difference between TENA and OASIS is that TENA focuses on interoperability between different application domains using CORBA, whereas OASIS focuses on separating and defining different concerns of software instrumentation so each can be implemented using domain-specific objects. Likewise, OASIS provides support for dynamic instrumentation and callback commands to software probes, which provides the foundation for software actuators.

VI. 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 pertaining to instrumenting DRE systems and determining what information 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 drawbacks, 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 early system lifecycle phases. 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:

- **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 whether they are present or not. Enterprise DRE system developers and testers thus have greater confidence they can incorporate instrumentation into production systems and still meet system QoS requirements. Our future work is investigating techniques to automate minimizing probe effects for dynamic instrumentation of enterprise DRE systems.
- **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 is investigating techniques to optimize OASIS's data collection, archiving, and retrieval capabilities by leveraging this metadata.
- **Data distribution services may improve publisher-subscriber architecture**. The current OASIS specification and architecture relies on CORBA to realize its publisher-subscriber requirements. A better solution, however, may be to use QoS-enabled publisher-subscriber middleware, such as the Data Distribution Service (DDS), to handle real-time notifications of collected metrics. Our future work is comparing DDS implementations against CORBA implementations to determine their relative value for OASIS.

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

REFERENCES

[1] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*. Wiley and Sons, New York, 2007.

[2] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture: Patterns and Pattern Languagesg, Volume 5*. Wiley and Sons, New York, 2007.

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

[4] 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.

[5] J. Chang, N. Park, and W. Lee. Adaptive selection of tuples over data streams for efficient load shedding. *International Journal of Computer Systems Science & Engineering*, 23(4):277–287, 2008.

[6] R. Feldt, R. Torkar, E. Ahmad, and B. Raza. Challenges with Software Verification and Validation Activities in the Space Industry. In *Third International Conference on Software Testing, Verification and Validation*, pages 225–234, April 2010.

[7] 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-1211-09, Indiana University-Purdue University Indianapolis, December 2009.

[8] 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.

[9] J. H. Hill, H. Sutherland, P. Staudinger, T. Silveria, D. C. Schmidt, J. M. Slaby, and N. A. Visnevski. OASIS: A Service-Oriented Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems. In *Proceedings of 13th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, Carmona, Spain, May 2010.

[10] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley and sons, 2004.

[11] O. Laurent. Using Formal Methods and Testability Concepts in the Avionics Systems Validation and Verification (V&V) Process. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, April 2010.

[12] D. A. Menasce, L. W. Dowdy, and V. A. F. Almeida. *Performance by Design: Computer Capacity Planning By Example*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[13] Microsoft Corporation. Microsoft .NET Framework 3.0 Community. www.netfx3.com, 2007.

[14] J. R. Noseworthy. Developing Distributed Applications Rapidly and Reliably Using the TENA Middleware. In *IEEE Military Communications Conference*, volume 3, pages 1507–1513, October 2005.

[15] J. R. Noseworthy. The Test and Training Enabling Architecture (TENA) Supporting the Decentralized Development of Distributed Applications and LVC Simulations. In *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 259–268, October 2008.

[16] J. R. Noseworthy. Supporting the Decentralized Development of Large-Scale Distributed Real-Time LVC Simulation Systems with TENA (The Test and Training Enabling Architecture). In *Distributed Simulation and Real Time Applications, IEEE/ACM International Symposium on*, pages 22–29, 2010.

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

[18] R. Obermaisser, C. El-salloum, B. Huber, and H. Kopetz. Modeling and Verification of Distributed Real-time Systems using Periodic Finite State Machines. 23, 2008.

[19] Object Management Group. *Data Distribution Service for Real-time Systems Specification*, 1.0 edition, Mar. 2003.

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

[21] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 edition, Jan. 2008.

[22] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 2: CORBA Interoperability*, OMG Document formal/2008-01-07 edition, Jan. 2008.

[23] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 edition, Jan. 2008.

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

[25] 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.

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

[27] S. Poulding and J. A. Clark. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Transactions on Software Engineering*, 36:763–777, 2010.

[28] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[29] 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.

[30] Sun Microsystems. Java^{TM} 2 Platform Enterprise Edition. java.sun.com/j2ee/index.html, 2007.

[31] 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.

[32] 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.

[33] P. Tröger and A. Polze. Object and process migration in .NET. 24, 2009.

[34] D. R. Wallace and R. U. Fujii. Software Verification and Validation: An Overview. *IEEE Software*, 6:10–17, 1989.