Experiences with Service-Oriented Middleware for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems

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

Douglas C. Schmidt
Carnegie Mellon University
Pittsburgh, PA USA
Email: dschmidt@sei.cmu.edu

***Abstract*—This paper describes our experiences applying a test and evaluation (T&E) middleware framework called the *Open-source Architecture for Software Instrumentation Systems (OASIS)* to the Unified SHIP platform, which is a representative system for next-generation shipboard computing systems. The OASIS middleware framework discussed in this paper enables instrumenting distributed real-time and embedded (DRE) systems to collect and extract metrics without *a priori* knowledge of the metrics collected. The flexibility of OASIS's metametrics-driven approach to instrumentation and data collection increased developer and tester knowledge and analytical capabilities of end-to-end QoS in shipboard computing systems.**

I. Introduction

Shipboard computing systems are a class of enterprise distributed real-time and embedded (DRE) systems with stringent quality-of-service (QoS) requirements (such as latency, response time, and scalability) that must be met in addition to functional requirements [1]. To ensure QoS requirements of such DRE systems, developers must analyze and optimize end-to-end performance throughout the software lifecycle. Ideally, this test and evaluation (T&E) [2] process should start in the architectural design phase of shipboard computing, as opposed to waiting until final system integration later in the lifecycle when it is more expensive to fix problems.

T&E of shipboard computing system QoS requirements typically employs software instrumentation techniques [1], [3]–[5] that collect metrics of interest (*e.g.*, CPU utilization, memory usage, response of received events, and heartbeat of an application) while the system executes in its target environment. Performance analysis tools then evaluate the collected metrics and inform system developers and testers whether the system meets its QoS requirements. These tools can also identify bottlenecks in system and application components that exhibit high and/or unpredictable resource usage [6], [7].

Although software instrumentation facilitates T&E of shipboard computing system QoS requirements, conventional techniques for collecting metrics are typically highly-coupled to the system's implementation [1], [2], [8]. For example, shipboard computing developers often decide during the system design phase what metrics to collect for T&E, as shown in Figure 1. Developers then incorporate into the system's design the necessary probes to collect these metrics from the distributed environment.

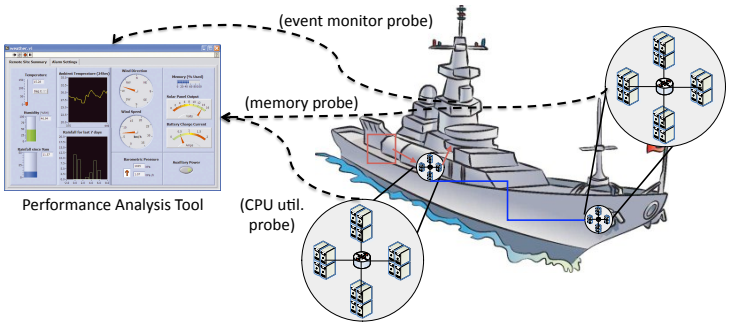

Figure 1. Conventional Way to Instrument Shipboard Computing Systems

The drawback with a tightly-coupled approach is that shipboard computing developers must either (1) redesign the system to incorporate the new/different metrics or (2) use *ad hoc* techniques, such as augmenting existing code with the necessary interfaces without understanding its impact to the overall system's design and maintainability, to collect such metrics. Developers therefore need better techniques to simplify instrumenting shipboard computing systems for collecting and extracting metrics—especially when the desired metrics are not known *a priori*.

The *Embedded Instrumentation Systems Architecture* (EISA) [9] initiative defines a *metadata-driven method* for heterogeneous data collection and aggregation in a synchronized and time-correlated fashion [9], as opposed to an *interface-centric method* [10] used in conventional DRE systems. Instead of integrating many interfaces and methods to extract and collect metrics into the system's design, EISA treats all metrics as arbitrary data that flows over a common reusable channel and discoverable via metametrics.[1] EISA thus helps reduce the coupling between system design and

[1]Metametrics are metadata that describe metrics collected at runtime without knowing its structure and quantity *a priori*.

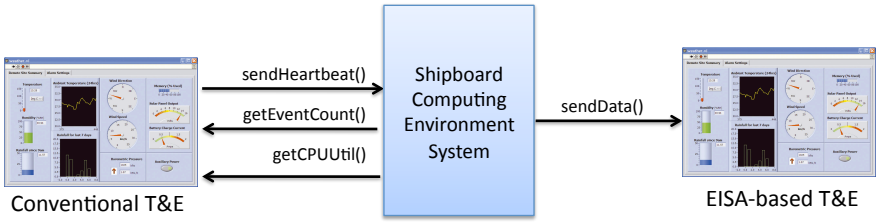

Figure 2. Conventional Approach vs. EISA's Approach to T&E

instrumentation logic incurred with the conventional T&E techniques described above [11], as shown in Figure 2.

Initial implementations of the EISA standard focused mainly on hardware instrumentation. To apply the EISA standard in the software domain, we developed the *Open-source Architecture for Software Instrumentation of Systems (OASIS)*. This experience report discusses our insights and lessons learned while developing and applying OASIS to a representative shipboard computing project. The main contributions of this experience report include:

- A discussion of design choices made while designing and implementing OASIS,
- An analysis of current limitations of the OASIS architecture, as well as insights on how such limitations can be addressed,
- A summary of open research challenges associated with instrumenting DRE systems.

Our experiences gained from developing and applying OASIS to shipboard computing show that EISA's metadata-driven approach to instrumentation and data collection provides flexibility that can increase DRE system developers and tester's knowledge base and analytical capabilities of end-to-end QoS. OASIS also provides a solid foundation for addressing open problems associated with instrumenting DRE systems.

Paper organization. The remainder of this paper is organized as follows: Section II provides an overview the representative shipboard computing system we use as a case study for our work, and of OASIS focusing on key instrumentation challenges; Section III describes how OASIS addresses these challenges; Section IV compares OASIS with related work; and Section V presents concluding remarks.

II. Case Study: The Unified SHIP Platform

EISA-based tools have primarily been used to instrument hardware components (*e.g.*, sensor hardware components) of DRE systems [9]. These systems, however, are composed of both hardware and software components. Ideally, end-to-end QoS evaluation of shipboard computing systems should employ performance analysis of both hardware and software components.

To help evaluate EISA in a representative enterprise DRE system, we created the *Unified Software/Hardware Instrumentation Proof-of-concept (Unified SHIP)* platform, which provides a representative environment for investigating technical challenges of next-generation shipboard computing systems. The Unified SHIP platform contains software components (*i.e.*, the rectangles in Figure 3) implemented using the Component Integrated ACE ORB (www.dre.vanderbilt.edu/CIAO), which is a C++ implementation of the Lightweight CORBA Component Model [12]. Likewise, performance analysis tools are implemented using a variety of programming languages, such as C++, C#, and Java. The software applications run on real-time Linux and Solaris operating systems, whereas performance analysis tools run on Windows and conventional Linux operating systems.

Figure 3 also shows how the Unified SHIP platform consists of EISA-compliant sensor hardware components and a collection of software components that performed the following operational capabilities for shipboard computing systems: 4 components are trackers that monitor events in the operational environment, 3 components are planners that process data from the sensor components, 1 component performs configuration of the effectors, 3 components are effectors that react to commands from the configuration component, 3 components allow operators to send commands to the planner components, and 1 component is a gateway that authenticates login credentials from the operator components. The directed line between each component in Figure 3 represents inter-component communication, such as sending an event between two different components.

Existing techniques for instrumenting shipboard computing systems assume software instrumentation concerns (*e.g.*, what metrics to collect and how to extract metrics from the system) are incorporated into the system's design. Since the Unified SHIP platform consists of hardware and software components at various degrees of maturity and deployment, it is hard to use existing instrumentation techniques to collect and extract metrics for QoS evaluation during early phases of the software lifecycle. In particular, developers and testers of the Unified SHIP platform faced the following challenges:

- **Challenge 1: Describing metametrics in a platform- and language-independent manner.** The heterogeity of the Unified SHIP platform's software and hardware components makes it undesirable to tightly couple performance analysis tools to the target platform and

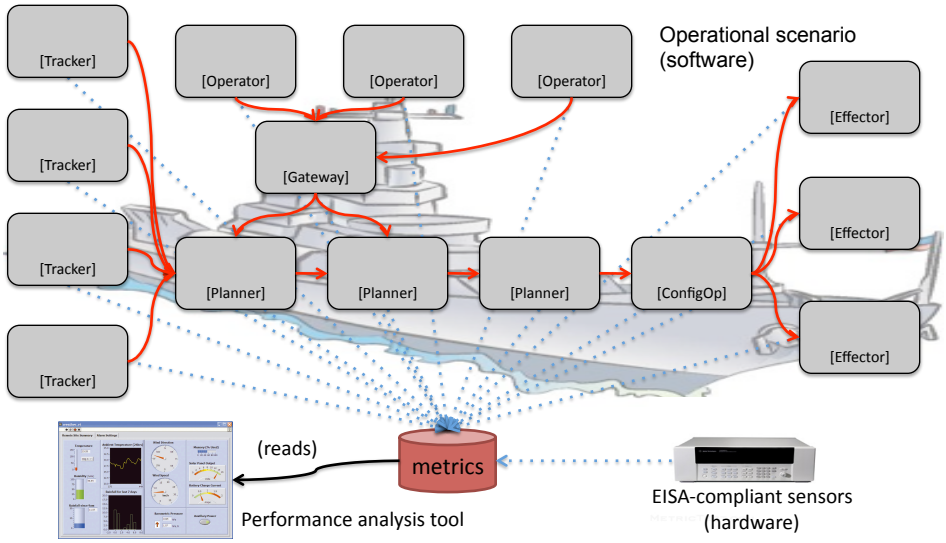

Figure 3. Overview of the Unified SHIP Platform

language of software and hardware components to collect and analyze metrics. Platform- and language-independent techniques and tools are therefore needed that will enable description of metrics collected from hardware and software components.

- **Challenge 2: Collecting metrics without a priori knowledge of its structure and quantity.** Metrics collected via instrumentation in the Unified SHIP platform come from heterogenous sources, which make it tedious and error-prone for system developers and testers to tightly couple the systems implementation to understand each metric and technology a priori. Techniques are therefore needed that will enable the collection of metrics from the Unified SHIP platform for QoS evaluation without a priori knowledge of which metrics are collected.

The remainder of this experience report discusses how different design choices in OASIS enabled us to address these two challenges in context of the Unified SHIP platform.

III. Experiences from Applying OASIS to the Unified SHIP Platform

This section discusses our experience applying OASIS to the Unified SHIP Platform introduced in Section II. For each experience discussed in this paper, we first introduce the experience topic and then give a detailed account of our experience—both positive and negative when applicable.

A. Brief Overview of OASIS

OASIS is dynamic instrumentation middleware for DRE systems that uses a metametics-driven design integrated with loosely coupled data collection facilities. Metametrics are defined as software probes, which are autonomous agents that collect both system and application-level metrics. Listing 1 highlights an example software probe—written in OASIS's *Probe Definition Language (PDL)*—that collects memory statistics. OASIS's PDL compiler uses such definitions to generate a stub, skeleton, and base implementation for the target programming language, and a *XML Schema Definition* (XSD) file that details the structure of a memory probe's data (see Figure 4). The stub is used in the *Performance Analysis Tool* (shown as PAT in Figure 4) to recall data, the skeleton and base implementation are used in the instrumented application (App. in Figure 4) to collect metrics, and the XSD file is used for dynamic discovery of metrics.

```
1  [uuid(ed970279-247d-42ca-aeaa-bef0239ca3b3); version(1.0)]
2  probe MemoryProbe {
3    uint64 total_physical_memory;
4    uint64 avail_physical_memory;
5    uint64 total_virtual_memory;
6    uint64 avail_virtual_memory;
7    uint64 cache;
8    uint64 commit_limit;
9    uint64 commit_total;
10 };
```

Listing 1. Definition of a memory probe in OASIS.

Figure 5 shows a high-level diagram of OASIS architecture and data collection facilities. As shown in this figure, this portion of OASIS consists of the following entities:

- **Embedded instrumentation node (EINode)**, which 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

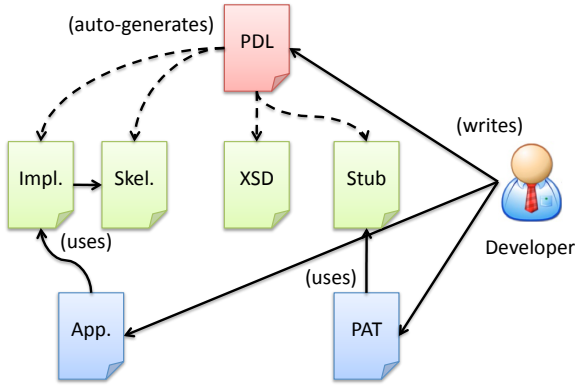

Figure 4. Overview of Files Generated from a PDL Probe by OASIS

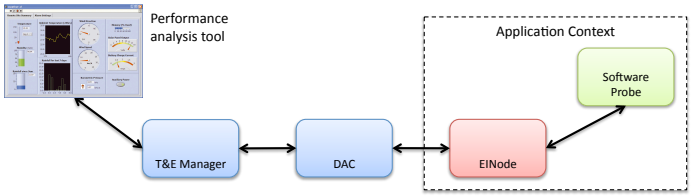

Figure 5. Architectural Overview of the OASIS Middleware

boundaries, only process 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 distinguished by a unique user-defined UUID and corresponding human-readable name.

- **Data acquisition and controller (DAC)**, which receives data from an EINode and archives it for acquisition by performance analysis tools, such as querying the performance of the latest state of component collected by a application-level software probe. The DAC is a persistent database with a consistent location in the target environment that can be located via a naming service. 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. The DAC registers itself when the test and evaluation manager (see below) when it is created and is identifiable by a unique user-defined UUID and corresponding human-readable name.
- **Test and Evaluation (T&E) manager** , which is the main entry point for user applications (see below) into OASIS. The T&E manager gathers data from each DAC that has registered with it. The T&E manager also enables user applications to send signals to each software probe in the system at runtime to alter its behavior, *e.g.*, by decreasing/increasing the hertz of the heartbeat software probe in the Unified SHIP platform scenario. This dynamic behavior is possible because the T&E manager is aware of all its DACs in the system, the DACs are aware of all its EINodes, and the EINodes are aware of all their registered software probes.
- **Performance analysis tools**, which are domain-specific tools, such as distributed resource managers and real-time monitoring and display consoles from the Unified SHIP platform, that interact with OASIS by requesting metrics collected from different software probes via the T&E manager. Tools can also send signals/commands to software probes to alter their behavior at runtime. This design enables 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.

Figure 6 shows the integration of OASIS with the Unified SHIP platform. Each hardware and software component is associated with an EINode that contains a set of software probes (or instruments in the case of hardware components [11]) that collect and submit metrics for extraction from the system. When an EINode receives metrics from a software probe (or instrument), it sends it to a DAC for storage and on-demand retrieval. Performance analysis tools then request collected metrics via the T&E manager, which locates the appropriate metrics in a DAC.

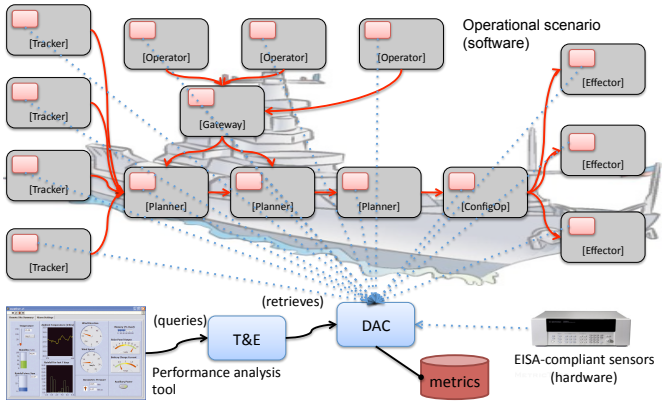

Figure 6. Integration of OASIS with the Unified SHIP Platform

Using this architecture, it is possible for the OASIS middleware framework to collect and analyze metrics without *a priori* knowledge of either the structure and complexity. The remainder of this section discusses how different design choices have impacted our experience using OASIS on the Unified SHIP Platform.

Experience 1: On Separating Metrics from Metametrics

In OASIS, metrics are separated from metametrics (*i.e.*, information that describes the metric's structure and types). The metametics are defined using XML Schema Definition (XSD) (see Listing 2 for an example), whereas metrics

are packaged as blobs of data. As shown in Figure 7, the software probes package the data, prepend a header, and pass the metrics to the EINode. The EINode then prepends its header information and forwards it to the DAC. During this packaging process, however, no metametrics are stored with the actual metrics. Instead, the metametrics are sent to the DAC for storage when an EINode registers itself with a DAC.

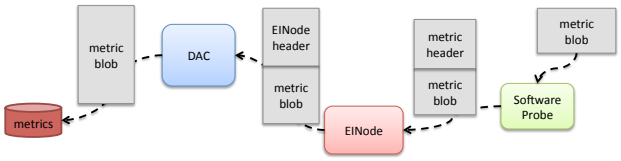

Figure 7. The Metric Collection and Packaging Process in OASIS

Based on our experience applying OASIS to the Unified SHIP platform, we learned that separating metrics from metametics has the following advantages:

A1. Portability across different architectures. For example, the Unified SHIP platform consists of many different middleware technologies, such as the Common Object Request Broker Architecture (CORBA) [10], [13], [14], the Data Distribution Services [15], and Microsoft .NET [16]. None of these technologies, however, provide a straightforward or standard method for discovering metametrics that is portable across programming languages, middleware technologies, and platforms.

Moreover, the technologies used in the Unified SHIP platform assume that communication occurs between two strongly-typed endpoints. For example, in CORBA the client and server use strongly-typed interfaces that know what data types are sent across the network. The `CORBA::Any` element type is used in CORBA to send data without *a priori* knowledge. This element type knows the data type (*e.g.*, `tk_long`, `tk_short`, and `tk_boolean`). It does not, however, know the structure of complex types (*e.g.*, `tk_struct`), which makes it hard for the DAC to store metrics in its database.

For example, there is no standard method for discovering a metrics structure or serializing it to a blob of data using the generic `CORBA::Any` type. In some programming languages, such as Java and C#, it is possible to use reflection to support this requirement. This approach is only possible, however, because metametrics are built into the programming language. The serialization problem can also be solved by forcing the DAC to know each kind of metrics collected by a software probe. When a new metric arrives at the DAC, the DAC locates a corresponding software probe stub that can serialize data contained in the generic type. This approach, however, requires the DAC to know *a priori* all the software probes used in the Unified SHIP platform, which is not possible since developers can add new probes as they identify more data types to instrument and collect.

A2. Self-containment for offline analysis. Another advantage of separating metrics from metametrics is self-contained storage for offline analysis of data since the DAC stores both metametrics and metrics for a given execution of the Unified SHIP platform in a single database. This database can then be archived and recalled later to compare results of different test executions of the Unified SHIP platform. Moreover, developers can create new analysis tools at later dates to analyze different aspects of the data.

In our experience applying OASIS to the Unified SHIP platform we have not yet found any disadvantages to separating metrics and metametrics. Its self-contained and standard method for storing and recalling metrics is platform-, language-, and technology-independent.

Experience 2: On Using XML Schema Definition to Describe Metametrics

Metametrics in OASIS are defined using XSD files (as shown in Listing 2).

```
1  <?xml version='1.0' ?>
2  <xsd:schema>
3   <xsd:element name='probeMetadata' type='stateType' />
4   <xsd:complexType name='stateType'>
5    <xsd:sequence>
6     <xsd:element name='component' type='xsd:string' />
7     <xsd:element name='state' type='xsd:integer' />
8    </xsd:sequence>
9   </xsd:complexType>
10 </xsd:schema>
```

Listing 2. An Example XML Schema Definition that Describes Component State Metrics Collected by a Software Probe.

When an EINode registers itself with the DAC, this information is sent to the DAC. The use of XSD to describe metametrics has the following advantage:

A3. GUI support. The main motivation for using XSD files to define metametics in OASIS is that there are existing tools that can create a *graphical user interface* (GUI) from a XSD file [17], which made it easier for Unified SHIP platform developers to visualize collected metrics as new software probes were added to the system. XSD is a verbose language since it is based on XML, *e.g.*, the metametrics in Listing 2 is approximately 300 bytes of data just to describe the metric's type name and its structure.

Using XSD to describe metametrics, however, has the following disadvantage:

D1. High processing overhead. Processing XSD files, which are XML files, can have high overhead and impact real-time performance. In OASIS, however, we do not process XSD files in real-time. Instead, they are processed at initialization time or when new metric types are discovered. Based on our experience with the Unified SHIP platform, the rate of discovering new metrics is not frequent enough to warrant using a less verbose method for defining metametrics—even when implementing generic performance analysis tools.

Experience 3: On Software Probe Definition and Stucture

Software probes in OASIS are defined using PDL. Developers define the metrics collected by a software probe, as shown in Listing 1 in the overview of OASIS. The OASIS compiler then generates the appropriate stubs and skeletons for using the software probe for instrumentation. The current implementation of OASIS does not support hierarchical software probe definitions, which means that each software probe definition is its own entity. This design choice, however, presented the following disadvantage:

D2. Lack of hierarchy increases instrumentation complexity. Based on our experience applying OASIS to the Unified SHIP platform, the lack of hierarchical software probe definitions increases the complexity of instrumenting such systems since developers must either:

- Define a software probe such that it is too broad in scope,
- Define a software probe that is too narrow in scope, or
- Create separate software probes that collect similar information with slight differences.

The problem with broad software probes is that they collect more information than is needed, *i.e.*, have fields that have no data on different platforms. Likewise, narrow software probes must sacrifice data in certain situations, such as not collecting a specific metric on the Linux platform since there is not an equivalent metrics on the Windows platform.

For example, in the Unified SHIP platform, software components execute on either a Windows or Linux platform. If developers want to collect memory metrics from either platform they would have to decide either to implement a broad or narrow software probe since each platform provides different information about memory usage, as shown in Table I. If a broad software probe were implemented the Unified SHIP platform developers would have to ensure that all metrics in Table I were covered. If a narrow software probe were implemented, conversely, they would only cover 8 common memory metrics (*i.e.*, MemTotal, MemFree, Cached, CommittedLimit, Committed_AS, VmallocTotal, VmallocUsed, and AvailVirtual), which also fails to account for mapping similar metrics to a common name and unit in the software probe's implementation.

Ideally, it should be possible for Unified SHIP platform developers to define hierarchical software probes to show relations between them. For example, Unified SHIP platform developers should be able to define a `MemoryProbe` that contains all metrics common across all platforms, as shown in Figure 8. Each specific platform-specific memory probe (*e.g.*, `LinuxMemoryProbe` and `WindowsMemoryProbe`) then extends the `MemoryProbe` definition, as needed.

Based on our needs, we have realized that supporting hierarchical software probe definitions, however, has the following advantages:

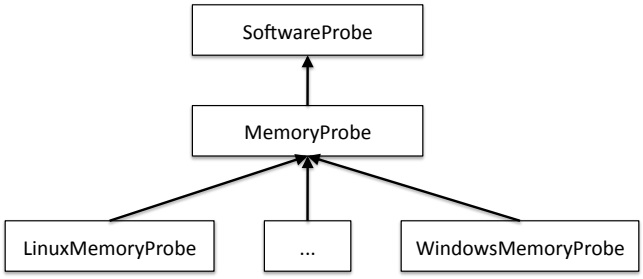

Figure 8. An Example of Hierarchically Defining the Memory Software Probe in OASIS

A4. Metric reuse. When we define software probes hierarchical as done in object-oriented programming languages, such as C++, C#, and Java, it allows similar software probes to reuse metric definitions. Unified SHIP Platform developers therefore need not make critical decisions as to whether they should implement broad or narrow software probes.

A5. Platform-specific vs. general-purpose performance analysis tools. OASIS allows performance analysis tools to request real-time updates when new data arrives. The hierarchical software probe definitions give performance analysis tools greater flexibility when registering for real-time updates. For example, they can request general memory probe data, *i.e.*, data collected by a probe of type `MemoryProbe`, or specific memory probe data, *i.e.*, either `WindowsMemoryProbe` or `LinuxMemoryProbe` data. The Unified SHIP platform developers can therefore implement general-purpose performance analysis tools or platform-specific performance analysis tools.

Experience 4: Observing Other Roles of the T&E Manager

The T&E Manager is the main entry point into the OASIS architecture for performance analysis tools, as described in Section II. This manager assists with gathering and correlating data requested by performance analysis tools. It also routes commands to software probes—via the DAC and EINode—to enable dynamic runtime behavior modifications, such as reducing its data collection frequency. Based on our experience applying OASIS to the Unified SHIP Platform, the T&E Manager has the following advantages:

A6. Domain-specific naming service. Based on our experience applying OASIS to the Unified SHIP platform, the T&E Manager is also a domain-specific naming service that keeps track of available DACs since the T&E manager must know all DACs available in test execution. Otherwise, it is hard for performance analysis tools to send commands to software probes. In addition, it is hard for performance analysis tools to register for real-time updates, which must be done by first locating an appropriate DACs via the T&E manager.

A7. Gateway and policy manager. Another role of the

Table I
COMPARISON OF MEMORY METRICS COLLECTED ON LINUX VS. WINDOWS PLATFORMS

Linux (/proc/meminfo)	Windows (Memory Performance Info [18])	Description
MemTotal	PhysicalTotal	Total amount of memory (avail. + used)
MemFree	PhysicalAvail	Total amount of memory free
Buffers		Amount of physical RAM used for file buffers
Cached	SystemCache	Amount of physical RAM used as cache memory
SwapCache		Amount of Swap used as cache memory
InActive		Total amount of buffer or page cache memory that are free and available
Active		Total amount of buffer or page cache memory, that is active
HighTotal		Total amount of memory in the high region
LowTotal		Total amount of non-highmem memory
LowFree		Amount of free memory of the low memory region
	KernelTotal	Sum of memory currently in paged and nonpaged kernel pools, in pages.
	KernelPaged	Memory currently in paged kernel pool, in pages.
	KernelNonpaged	Memory currently in nonpaged kernel pool, in pages.
	PageSize	Size of a page, in bytes
SwapTotal		Total amount of physical swap memory
SwapFree		Total amount of swap memory free
Dirty		The total amount of memory waiting to be written back to the disk.
WriteBack		The total amount of memory actively being written back to the disk
	CommitPeak	Max number of pages simultaneously in committed state
CommittedLimit	CommitLimit	Max memory available without extending paging files
Committed_AS	CommitTotal	Number of pages currently committed by the system
VmallocTotal	TotalVirtual	Total size of vmalloc memory area
VmallocUsed	TotalVirtual - AvailVirtual	Amount of virtual memory used
VmallocTotal - VmallocUsed	AvailVirtual	Amount of virtual memory available for allocation
VmallocChunk		Largest contiguous block of virtual memory that is free

T&E Manager that we learned is that it can be a gateway/-policy manager. In the Unified SHIP platform, some metrics collected by software probes should not be available to all performance analysis tools. For example, software metrics that would be considered sensitive metrics should not be available to performance analysis tools that do not have the correct privileges. The T&E Manager can therefore enforce such policies. Realizing this role of the T&E Manager also requires security enhancements at the DAC since metrics are stored in a database therefore for offline processing.

There is, however, a disadvantage to observing other roles of the T&E manager:

D3. The "god" T&E manager. If done incorrectly, the T&E manager could become a "god' T&E manager.[2] This superordination occurs when all roles of the T&E manager are condensed into a single entity, instead of decomposing it into distinct entities. We can overcome this design challenge via the Component Configurator [19] pattern, where each role is realized as a dynamically loadable component. The T&E manager then loads different components/roles as needed, ensuring the T&E manager is as lightweight as possible.

IV. RELATED WORK

This section compares our work on OASIS with related work.

[2]This name is derived from the "god" class [6] software performance antipattern where a single class contains all functionality, instead of modularizing it into a family of related classes.

Dynamic binary instrumentation (DBI) frameworks. Pin [20] and DynamoRIO [21] are examples of DBI frameworks. Unlike OASIS, both Pin and DynamoRIO do not require modification of existing source code to enable instrumentation. Instead, software developers use Pin to execute the application, and during the process Pin inserts points of instrumentation based on C/C++ user-created instrumentation tools—similar to performance analysis tools in OASIS. Although DBI frameworks address different problems, we believe they can work together in that software probes can be implemented as third-party analysis tools for DBI frameworks. This combination would allow OASIS to collect instrumentation information from the DBI framework that instruments a DRE system in real-time without modifying any of the existing source code—as done traditionally with OASIS.

DTrace [4] is another DBI framework. Unlike Pin and DynamoRIO, DTrace provides a scripting language for writing performance analysis tools. DTrace also has the ability to write custom software probes, which can be easily integrated into DTrace's collection facilities. DTrace's software probe design is therefore similar to OASIS in that it is extensible without *a priori* knowledge. It differs in that software metrics cannot be extracted from the host machine where software instrumentation is taking place.

Distributed data collection. Distributed Data Collector (DDC) [22] is a framework for collecting resource metrics, such as CPU and disk usage, from Windows personal computers (PCs). In DDC, software probe metrics are collected from PCs and stored in a central location. Unlike OASIS,

each software probe's metrics are stored its own file, which is then parsed by analysis tools. OASIS improves upon this design by storing all metrics in a single database, instead of separate files. Likewise, OASIS's data collection framework is platform-, language-, and architecture-independent (*i.e.*, not bound to only Windows PCs and Windows-specific software probes).

General-purpose middleware solutions can be used for distributed data collection. For example, the DDS is an event-based middleware specification that treats data as first-class entities. This concept is similar to OASIS in that events are similar to software probe metrics. The main difference is that DDS is a strongly-typed middleware solution in that both endpoints know the data type *a priori*. Moreover, there is not standard way to serialize the data in a DDS event. This therefore makes it hard to store metrics in the DAC's database.

V. Concluding Remarks

Test and evaluation (T&E) of shipboard computing system QoS during early phases of the software lifecycle helps increase confidence that the system being developed will meet it functional and QoS requirements. Conventional T&E instrumentation mechanisms, however, are tightly coupled with the system's design and implementation. This experience report therefore described how design choices in OASIS's implementation of the EISA standard helped reduced these coupling concerns. In addition, it also highlighted several revelations about different design choices that are currently being addressed in the OASIS middleware framework.

Based on our experience with OASIS, we found the following open research challenges, which extend the research directions presented in prior work [23], remain when instrumenting DRE systems:

- **Complex Event Processing.** Complex event processing [24] involves processing many different events and streams of data, across all layers of a domain, identifying meaningful events, and determining their impact on a given concern, such as performance, functionality, and scalability. Each software probe in OASIS can be viewed as stream of data and the DAC can be viewed as a data source with historical records. Likewise, performance analysis tools can register for real-time delivery of software probe data.
  Future research directions therefore include implementing complex event processing support in OASIS. Adding this support will be hard because the traditional use of complex event processing engines involves viewing results via a graphical user interface, which is considered one form of a performance analysis tool in OASIS. In reality, many different performance analysis tools (such as real-time monitoring and feedback performance analysis tools) should be able to leverage complex event processing support. Likewise, complex event processing has not been applied to general-purpose instrumentation middleware for DRE systems.
- **Data Fusion and Data Integration.** Data fusion [25] is the process of combining data from multiple sources for inference. The motivation for data fusion is that multiple data sources will be more accurate that a single data source. Although data fusion can be used to support complex event processing, it is a separate research area. Data integration [26], however, is the process of combining data from different sources to provide a unified view.
  When we examine the OASIS middleware framework, and each of its entities that play a major role in collecting and storing data (*e.g.*, software probe, EINode, and DAC), it is clear that data fusion and data integration techniques can be applied readily. The challenge, however, is understanding how both data fusion and data integration can be integrated with the real-time aspects of OASIS. Future research directions therefore involves addressing these challenges in OASIS so we can provide a general-purpose middleware solution for data fusion and data integration in OASIS.

As we apply OASIS to other application domains, such as resource-constrained embedded systems and mobile devices, we will continue identifying new research challenges. Since OASIS is an open-source middleware framework, it provides an effective foundation for ensuring that solutions to these open research challenges will be available to the T&E community.

OASIS is currently integrated into CUTS and is freely available for download in open-source format from cuts.cs.iupui.edu.

References

[1] Z. Tan, W. Leal, and L. Welch, "Verification of Instrumentation Techniques for Resource Management of Real-time Systems," *J. Syst. Softw.*, vol. 80, no. 7, pp. 1015–1022, 2007.

[2] G. Hudgins, K. Poch, and J. Secondine, "The Test and Training Enabling Architecture (TENA) Enabling Technology For The Joint Mission Environment Test Capability (JMETC) and Other Emerging Range Systems," in *Proceeding of U.S. Air Force T&E Days*, 2009.

[3] 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*, 1994, pp. 196–205.

[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*, June 2004, pp. 15–28.

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

[6] C. Smith and L. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Boston, MA, USA: Addison-Wesley Professional, September 2001.

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

[8] D. G. Waddington, N. Roy, and D. C. Schmidt, "Dynamic Analysis and Profiling of Multi-threaded Systems," in *Designing Software-Intensive Systems: Methods and Principles*, P. F. Tiako, Ed. Idea Group, 2007.

[9] N. Visnevski, "Embedded Instrumentation Systems Architecture," in *Proceedings of IEEE International Instrumentation and Measurement Technology Conference*, May 2008.

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

[11] A. Stefani and M. N. Xenos, "Meta-metric Evaluation of E-Commerce-related Metrics," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 233, pp. 59–72, 2009.

[12] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.

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

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

[15] *Data Distribution Service for Real-time Systems Specification*, 1.2 ed., Object Management Group, Jan. 2007.

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

[17] V. Radha, S. Ramakrishna, and N. kumar, "Generic XML Schema Definition (XSD) to GUI Translator," *Distributed Computing and Internet Technology*, pp. 290–296, 2005.

[18] M. Library, "Memory Performance Information," msdn.microsoft.com/en-us/library/aa965225\%28v=vs.85\%29.aspx.

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

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *SIGPLAN Notes*, vol. 40, pp. 190–200, June 2005.

[21] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275. [Online]. Available: http://portal.acm.org/citation.cfm?id=776261.776290

[22] P. Domingues, P. Marques, and L. Silva, "Distributed Data Collection through Remote Probing in Windows Environments," in *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*. IEEE, 2005, pp. 59–65.

[23] J. H. Hill, H. Sutherland, P. Staudinger, T. Silveria, D. C. Schmidt, J. M. Slaby, and N. Visnevski, "OASIS: An Architecture for Dynamic Instrumentation of Enterprise Distributed Real-time and Embedded Systems," *International Journal of Computer Systems Science and Engineering, Special Issue: Real-time Systems*, April 2011.

[24] P. Dekkers, "Complex Event Processing," Master's thesis, Radboud University Nijmegen, Nijmegen, Netherlands, October 2007.

[25] J. Bleiholder and F. Naumann, "Data Fusion," *ACM Computing Surveys*, vol. 41, pp. 1:1–1:41, January 2009. [Online]. Available: http://doi.acm.org/10.1145/1456650.1456651

[26] M. Lenzerini, "Data Integration: A Theoretical Perspective," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ser. PODS '02. New York, NY, USA: ACM, 2002, pp. 233–246. [Online]. Available: http://doi.acm.org/10.1145/543613.543644