Component-based System Integration via (Meta)Model Composition

Blinded for Review Purposes

Abstract— **This paper provides three contributions to the study of functional integration of distributed enterprise systems. First, we describe the challenges associated with functionally integrating the software of these systems. Second, we describe how the composition of domain-specific modeling languages (DSMLs) can simplify the functional integration of enterprise distributed systems by enabling the combination of diverse middleware technologies. Third, we demonstrate how composing DSMLs can solve functional integration problems in an enterprise distributed system case study by reverse engineering an existing CCM system and exposing it as Web Service(s) to web clients who use these services. This paper shows that functional integration done using (meta)model composition provides significant benefits with respect to automation, reusability, and scalability compared to conventional integration processes and methods.**

Index Terms— **Integration Engineering, Model-Driven Engineering, Component-based Systems, (Meta)Model Composition**

I. Introduction

A. Challenges of Functional Integration

With the emergence of commercial-off-the-shelf (COTS) component middleware technologies, such as Enterprise Java Beans (EJB) [1], CORBA Component Model (CCM) [2], and Microsoft .NET Framework [3], software developers are increasingly faced with the task of integrating heterogeneous enterprise distributed systems built using different COTS technologies, rather than just integrating proprietary software developed in-house. Although there are well-documented patterns [4] and techniques [5] for system integration using various component middleware technologies, system integration is still largely a tedious and error-prone manual process. To improve this process, component developers and system integrators must therefore understand key properties of the systems[1] they are integrating, as well as the integration technologies they are applying.

This paper describes technologies that help simplify the *functional integration* of systems built using component middleware. This type of integration operates at the logical business layer, typically using distributed objects/components, exposing existing functionality as services, or using messaging middleware. Functional integration of systems is hard due to the variety of available component middleware technologies. These technologies differ in many ways, including the protocol level, the data format level, the implementation language level, and/or the deployment environment level. In general, however, component middleware technologies are a more effective technology base than the brittle proprietary infrastructure used in legacy systems, which have historically been built in a vertical, stove-piped fashion.

Despite the benefits of component middleware, key challenges in functional integration remain unresolved when integrating large-scale systems developed using heterogeneous COTS middleware. These challenges include (1) *integration design*, which involves choosing the right abstraction for integration, (2) *interface mapping*, which reconciles different datatypes, (3) *technology mapping*, which reconciles various low-level issues, (4) *deployment mapping*, which involves planning the deployment of heterogeneous COTS middleware, and (5) *portability incompatibilities* between different implementations of the same middleware technology. The lack of simplification and automation in resolving these challenges significantly hinders effective system integration.

B. Solution Approach→Functional Integration using (Meta)Model Composition

A promising approach to address the integration challenges outlined above is *Model-Driven Engineering* (MDE), which involves the systematic use of models as essential artifacts throughout the software lifecycle [6]. At the core of MDE is the concept of *domain-specific modeling languages* (DSMLs) [7], whose type systems formalize the application structure, behavior, and requirements within particular domains, such as software defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of middleware platforms. DSMLs are described using *metamodels*, which define the relationships among concepts in a domain and precisely specify the key semantics and constraints associated with these domain concepts. Developers use DSMLs to build applications using elements of the type system captured by metamodels and express design intent declaratively rather than imperatively.

While DSMLs have been used to help software developers create homogeneous systems [8], [9], enterprise distributed systems are rarely homogeneous. A single DSML developed for a particular component middleware technology, such as EJB or CCM, may therefore not be applicable to model, analyze, and synthesize key concepts of Web Services. To integrate heterogeneous systems successfully, therefore, system integrators need tools that can provide them with a unified view of the entire enterprise system, while also allowing them fine-grained control over specific subsystems and components.

Our approach to integrating heterogeneous systems is *(meta)model composition*[2] [10], which (1) creates a new DSML from multiple existing DSMLs by adding new elements or extending elements of existing DSMLs, (2) specifies new relationships between existing elements, and (3) defines relationships between new and existing elements. A key benefit of (meta)model composition is its ability to add new capabilities while simultaneously leveraging prior investments in existing

[1] In the remainder of this paper, "system" refers to an enterprise distributed system built using component middleware like EJB, Microsoft .NET, or CCM.

[2] The term "(meta)model" conveys the fact that this composition technique can be applied to both metamodels *and* models.

tool-chains, including domain constraints and generators of existing DSMLs. A combination of DSMLs and DSML composition technologies can therefore help address the challenges outlined in Section I-A that are associated with functional integration of component middleware technologies, without incurring the drawbacks of conventional approaches.

This paper describes *System Integration Modeling Language* (SIML), which is our open-source DSML that enables functional integration of component-based systems via *(meta)model composition*. We developed SIML using the Generic Modeling Environment (GME) [11], which is an open-source meta-programmable modeling environment. SIML is a composite DSML that combines two existing DSMLs: (1) the CCM profile of the *Platform-Independent Component Modeling Language* (PICML),which supports model-driven engineering of CCM systems, and (2) the Web Services Modeling Language (WSML), which supports model-driven engineering of Web Services systems. Since SIML is a composite DSML, it has complete access to the semantics of PICML and WSML (sub-DSMLs), which simplifies and automates various tasks associated with integrating systems built using CCM and Web Services.

The remainder of this paper is organized as follows: Section II describes an enterprise distributed system case study built using component middleware that we used to evaluate functional integration technologies; Section III describes the DSML composition framework provided by GME to simplify the integration of heterogeneous systems; Section IV describes SIML, which uses GME's DSML composition framework to integrate heterogeneous enterprise distributed systems; Section V evaluates various approaches to system integration; and Section VI presents concluding remarks.

II. Functional Integration Case Study

To motivate the need for model-driven functional integration capabilities, this section describes an enterprise distributed system case study from the domain of *shipboard computing environments* [12], focusing on its functional integration challenges. A shipboard computing environment is a metropolitan area network of computational resources and sensors that provides on-demand situational awareness and actuation capabilities for human operators, and responds flexibly to unanticipated runtime conditions. To meet such demands in a robust and timely manner, the shipboard computing environment uses services to

- Bridge the gap between shipboard applications and the underlying operating systems and middleware infrastructure and
- Support multiple QoS requirements, such as survivability, predictability, security, and efficient resource utilization.

The shipboard computing environment that forms the basis for our case study was originally developed using one component middleware technology (OMG CCM) and was later enhanced to integrate with components written using another middleware technology (Web Services).

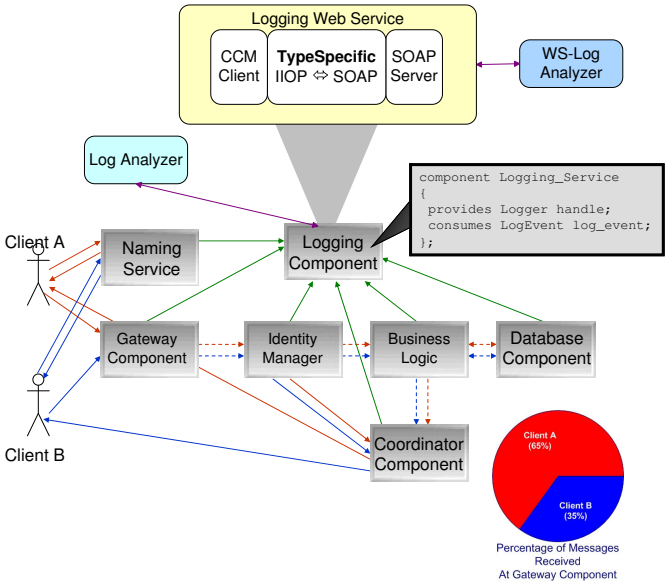

Fig. 1. **Enterprise Distributed System Architecture**

A. Architecture of the Case Study

The system in this case study consists of the following components, which are shown in Figure 1:

- **Gateway component**, which provides the user interface and main point of entry into the system for operators,
- **Naming Service components**, which are repositories that hold locations of services available within the system,
- **Identity Manager components**, which are responsible for user authentication and authorization,
- **Business logic components**, which are responsible for implementing the business logic,
- **Database components**, which are responsible for database transactions,
- **Coordinator components**, which act as proxies for "business logic" components and interact with clients,
- **Logging components**, which are responsible for collecting log messages sent by other components,
- **Log Analyzer components**, which analyze logs collected by Logging components and display results.

Clients that use the component services outlined above first connect to a Naming Service to obtain the Gateway's location. They then request services offered by the system, passing their authentication/authorization credentials to a Gateway component, which initiates the series of interactions shown in Figure 1. Depending on the credentials supplied by clients the system provides differentiated services. Areas where services can be differentiated between various clients include the maximum number of simultaneous connections, maximum amount of bandwidth allocated, and maximum number of requests processed in a particular time period.

To track the performance of the system—and the quality of service (QoS) the system offers to different clients—developers originally wrote Log Analyzer components to obtain information by analyzing the logs. Based on changes in the COTS technology base and user demand, the decision was

made to expose a Web Service API to Logging components so that clients could also track the QoS provided by the system to their requests by accessing information available in Logging components. Since the original system was written using CCM there was a new requirement to integrate systems that were not designed to work together, *i.e.*, CCM-based Logging components with the Web Service clients.

The flow of control, and the number and functionality of the different participants, in this case study is representative of shipboard systems that require authentication and authorization from clients, and provide differentiated services to clients, based on the credentials offered by the client. Below, we examine this system from an *integration* perspective, *i.e.*, how can this system—which was not designed for integration with other middleware—be integrated with other middleware.[3]

B. Functional Integration Challenges

Functional integration of systems is hard and involves activities that map between various levels of abstraction in the integration lifecycle, including design, implementation, and use of tools. We now describe key challenges associated with integrating older component middleware technologies, such as CCM and EJB, with newer technologies, such as Web Services, and relate them to our experiences developing the shipboard computing case study described in Section II-A.

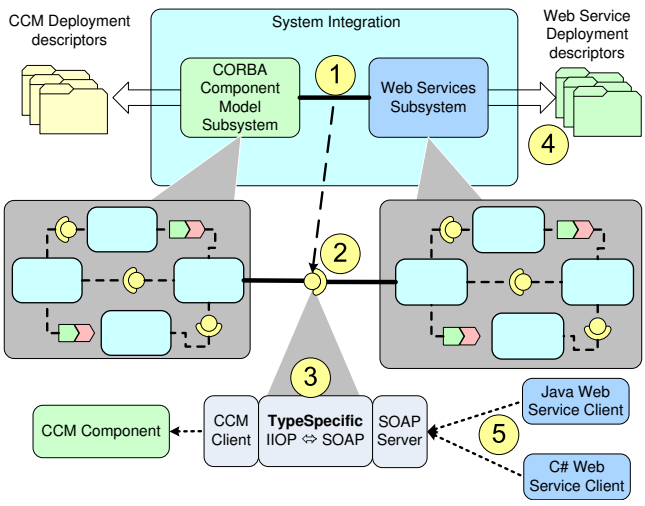

Fig. 2. **Functional Integration Challenges**

Challenge 1. Choosing an appropriate level of integration. As shown in Step 1 of Figure 2, a key activity is to identify the right level of abstraction at which functional integration should occur, which involves selecting elements from different technologies being integrated that can serve as conduits for exchanging information. Attempting integration at the wrong level of abstraction can yield brittle integration architectures that break when changes occur to either the source or target system being integrated.

In our case study example, we need to integrate Logging components such that Web Service clients can access their services. The programming model of CCM prescribes component *ports* as the primary component interconnection mechanism. Web Services also defines ports as the primary interconnection mechanism between a Web Service and its clients. During functional integration of CCM with Web Services, therefore, a mapping between CCM component ports and Web Services ports offers an appropriate level of abstraction for integration. In general, it is hard for system integrators to decide the right level of abstraction, and requires expertise in all of the technologies being integrated.

Challenge 2. Reconciling differences in interface specifications. After the level of abstraction to perform functional integration is determined, it is necessary to map the interfaces exposed by elements of the different technologies as shown in Step 2 of Figure 2. Common COTS middleware technologies usually have an interface definition mechanism that is separate from the implementation details, *e.g.* CCM uses the OMG Interface Definition Language (IDL), whereas Web Services use W3C Web Services Definition Language (WSDL). Irrespective of the mechanism used to define interfaces, mapping of interfaces between any two technologies involves at least three tasks: (1) *datatype mapping*, which involves mapping a datatype (both pre-defined and complex types) from source to target technology, (2) *exception mapping*, which involves mapping exceptions from source to target technology; exceptions are not clubbed together with datatypes since the source or target technologies might not have a notion of exceptions (*e.g.* Microsoft's COM uses a *HRESULT* to convey errors instead of using exceptions), and (3) *language mapping*, which involves mapping datatypes between two technologies while accounting for differences in languages at the same time.[4]

In our case study example, Logging components handle CORBA datatypes, whereas Web Service clients exchange XML datatypes. Performing these mappings is non-trivial, requires expertise in both the source and target technologies, and exposes severe scalability problems due to their tedium and error-proneness if they are not automated.

Challenge 3. Managing differences in implementation technologies. The interface mapping described above addresses the high-level details of how information is exchanged between different technologies being integrated. As shown in Step 3 of Figure 2, however, low-level technology details such as networking, authentication and authorization *et al.* are responsible to actually delivering such integration. This involves a technology mapping and includes the following activities: (1) *protocol mapping*, which reconciles the differences between the protocols used for communication between the two technologies, (2) *discovery mapping*, which allows bootstrapping and discovery of components/services between source and target technologies, and (3) *QoS mapping*, which maps QoS mechanisms between source and target technologies to ensure that service-level agreements (SLAs) are maintained.

In our case study example, Logging components only understand IIOP, which is completely different from the SOAP protocol understood by Web Service clients. While the Logging component is exposed to clients as a CORBA Object

[3]Note that this paper is *not* studying the system from the perspective of application functionality or the QoS provided by *Business Logic* components.

[4]Functional integration is very limited when attempting the latter mapping, which is often done via inter-process communication.

Reference registered with a Naming Service, a Web Service client typically expects a Uniform Resource Identifier (URI) registered with a Universal Description Discovery and Integration (UDDI) service, to indicate where it can obtain a service. Mapping of protocol, discovery, and QoS technology details requires not only expertise in the source/target technologies, but also intimate knowledge of the implementation details of these technologies.

Challenge 4. Managing deployment of subsystems. Component middleware technologies use declarative notations (such as XML descriptors, source-code attributes, and annotations) to capture various configuration options. Example metadata include EJB deployment descriptors, .NET assembly manifests, and CCM deployment descriptors. As shown in Step 4 of Figure 2, system integrators must track and configure metadata correctly during integration and deployment. In many cases, the correct functionality of the integrated system depends on correct configuration of the metadata.

In our case study example, Logging components are (1) associated with CCM descriptors needed to configure their functionality, (2) deployed using the CCM deployment infrastructure, and (3) run on a dedicated network testbed. If Web Service clients need to access functionality exposed by Logging components, however, certain services (such as a Web Server to host the service and a firewall) must be configured. This coupling between the deployment information of Logging components and the services exposed to Web Service clients means that changes to Logging component necessitates corresponding changes to Logging Web Service. Failure to keep these elements in sync usually results in loss of service to clients of one or both technologies.

Challenge 5. Dealing with interoperability issues. Unless a middleware technology has only one version implemented by one provider, there may be multiple implementations from different providers. As shown in Step 5 of Figure 2, differences between these implementations will likely arise due to non-conformant extension to standards, different interpretations of the same (often vague) specification, or implementation bugs. Regardless of the reasons for incompatibility, however, problems arise that often manifest themselves only during system integration. Examples of such differences are highlighted by the presence of efforts like the Web Services-Interoperability Basic Profile (WS-I) [13], which is a standard aimed at ensuring compatibility between the Web Services implementations from different vendors.

In our case study example, not only do Logging components need to expose their services in WSDL format, they must also ensure that Web Service clients developed using different Web Services implementations (*e.g.*, Microsoft .NET vs. Java) are equally able to access their services. Logging components therefore need to expose their services using an interoperable subset of WSDL defined by WS-I, so clients are not affected by incompatibilities, such as using SOAP RPC encoding.

Due to the challenges described above, significant integration effort is spent on configuration activities, such as modifying deployment descriptors, and interoperability activities, such as handcrafting protocol adapters to link different systems together, which does not scale up as the number of components in the system increases or the number of adaptations required increases. Problems discovered at integration stage often require changes to the implementation, and thus necessitate interactions between developers and integrators. These interactions are often inconvenient, and even infeasible (especially when using COTS products), and can significantly complicate integration efforts. The remainder of this paper shows how our GME-based (meta)model composition framework and associated tools help address these challenges.

III. DSML Composition using GME

This section describes the (meta)model composition framework in the Generic Modeling Environment (GME) [11]. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is *meta-programmable*, it can be used to design DSMLs, as well as build models that conform to a DSML.

DSMLs are defined by metamodels, hence, DSML composition is defined by (meta)model composition. The specification of how metamodels should be composed, *i.e.*, what concepts in the metamodels that are composed relate to each other and how, can be specified via normal association relationships and additional composition operators, as described in GME [10].

A key property of a composite DSML is that it supports the *open-closed* principle [14], which states that a class should be open for extension but closed with respect to its public interface. In GME, elements of the sub-DSMLs are *closed*,*i.e.*, their semantics cannot be altered in the composite DSML. The composite DSML itself, however, is *open*, *i.e.*, it allows the definition of new interactions and the creation of new derived elements. All tools that are built for each sub-DSML work without any modifications in the composite DSML and all the models built in the sub-DSMLs are also usable in the composite DSML.

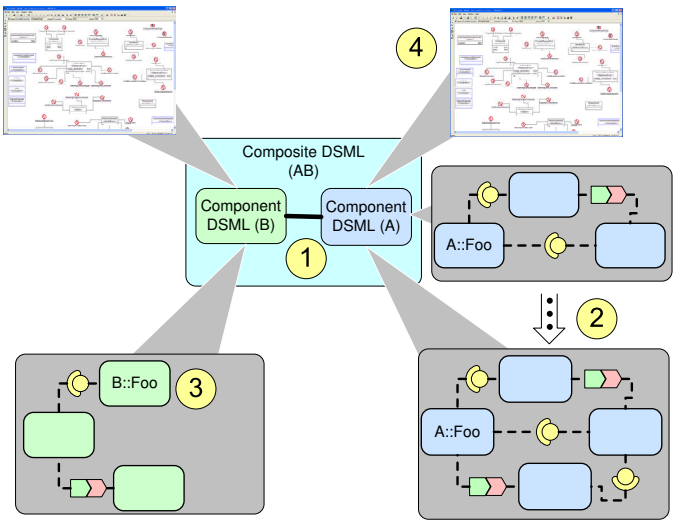

Fig. 3. **Domain-Specific Modeling Language Composition in GME**

We use the following GME (meta)model composition features to support the SIML-based integration of systems built using different middleware technologies, as described in Section IV:

• **Representation of independent concepts.** To enable complete reuse of models and tools of the sub-DSMLs, the composition must be done in such a way that all concepts defined in the sub-DSMLs are preserved. As shown in Step 1 of Figure 3, no elements from either sub-DSMLs should be merged together in the composite DSML. GME's composition operators [10] can be used to create new elements in the composite DSML, but the sub-DSMLs as a whole must remain untouched. As a consequence, any model in a sub-DSML can be imported into the composite language, and vice versa. All models in the composite language that are using concepts from the sub-DSMLs can thus be imported back into the sub-DSML. Existing tools for sub-DSMLs can be reused as well in the composite environment. This technique of composing DSMLs is referred to as *metamodel interfacing* [15] since we create new elements and relationships that provide the interface between the sub-DSMLs.

• **Supporting (meta)model evolution.** DSML composition enables reuse of previously defined (sub-)DSMLs. Just like code reuse in software development, (meta)model reuse can also benefit from the concept of libraries. If an existing (meta)model is simply copied into new composite (meta)models, any changes or upgrades to the original will not propagate to the places where they are used. As shown in Step 2 of Figure 3, if the original (meta)model is imported as a library, GME provides seamless support to update it when new versions become available (libraries are supported in any DSML with GME, not just the metamodeling language) Libraries are read-only projects imported to a host project. Components in the host project can create references to and derivations of library components. The library import process creates a copy of the reused project, so subsequent modifications to the original project are not updated automatically. To update a library inside a host project, a user-initiated refresh operation is required. To achieve unambiguous synchronization, elements inside a project have unique ids, which facilitates correct restoration of all relationships that are established among host project components and the library elements.

• **Partitioning (meta)model namespaces.** When two or more (meta)models are composed, name clashes may occur. To alleviate this problem, (meta)model libraries (and hence the corresponding components DSMLs) can have their own namespaces specified by (meta)modelers, as shown in Step 3 of Figure 3. External software components, such as code generators or model analysis tools that were developed for the composite DSML, must use the fully qualified names. But tools that were developed for component DSMLs will still work because GME sets the context correctly before invoking such a component.

• **Handling constraints.** The syntactic definitions of a metamodel in GME can be augmented by static semantics specifications in the form of Object Constraint Language (OCL) constraint expressions. When metamodels are composed together, the predefined OCL expressions coming from a sub-DSML should not be altered. Therefore GME's Constraint Manager uses namespace specifications to avoid any possible ambiguities, and these expressions are evaluated by the Constraint Manager with the correct types and priorities as defined by the sub-DSML as shown in Step 4 of Figure 3. The composite DSML can also define new OCL expressions to specify the static semantics that augment the specifications originating in the metamodels of the sub-DSMLs.

IV. Integrating Systems with SIML

This section describes how we created and applied the *System Integration Modeling Language* (SIML), which is our open-source composite DSML that simplifies functional integration of component-based systems built using heterogeneous middleware technologies.

A. The Design and Functionality of SIML

Applying GME's (meta)model composition features to SIML. To support integration of systems built using different middleware technologies, SIML uses the GME (meta)model composition features described in Section III and shown in Figure 4. SIML is thus a composite DSML that allows

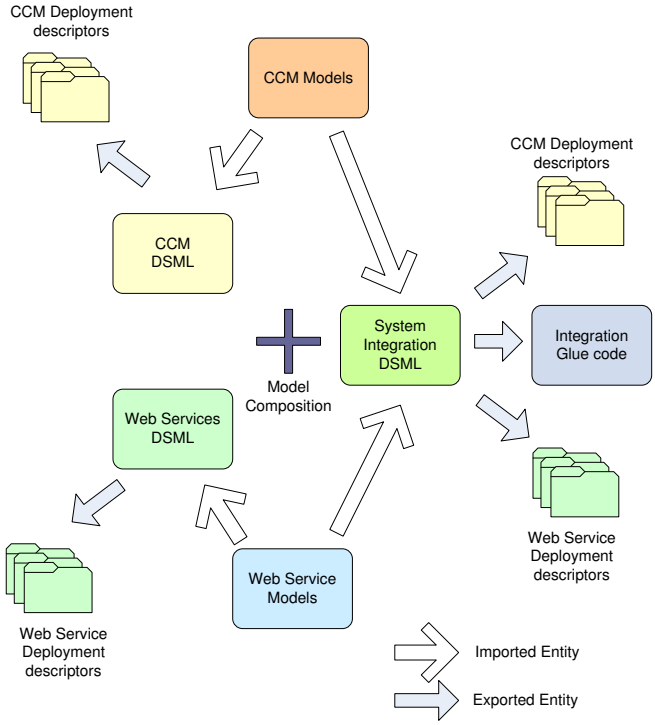

Fig. 4. **Design of System Integration Modeling Language**

integration of systems by composing multiple DSMLs, each representing a different middleware technology. Each sub-DSML is responsible for managing the metadata (creation, as well as generation) of the middleware technology it represents. The composite DSML defines the semantics of the integration, which might include reconciling differences between the diverse technologies, as well as representing characteristics of various implementations. System integrators therefore have a single environment that allows the creation and specification of elements in each sub-DSML, as well as interconnecting them as if they were elements of a single domain.

Applying SIML to compose CCM and Web Services. Our initial use of SIML was to help integrate CCM with Web

Services in the context of the shipboard computing case study described in Section II. The two sub-DSMLs we needed to integrate to support the new requirements described in Section II were:

- The **Platform-Independent Component Modeling Language** (PICML), which enables developers of CCM-based systems to define application interfaces, QoS parameters, and system software building rules, as well as generate valid XML descriptor files that enable automated system deployment.
- The **Web Services Modeling Language** (WSML), which enables development of Web Services, and supports key activities in Web Service development, such as creating a model of a Web Services from existing WSDL files, specifying details of a Web Service including defining new bindings, and auto-generating artifacts required for Web Service deployment.

The case study described in Section II provided the motivation to integrate them together using GME's (meta)model composition framework.

Since SIML is a composite DSML, all valid elements and interactions from both PICML and WSML are valid in SIML. It is therefore possible to design both CCM components (and assemblies of components), as well as Web Services (and federations of Web Services) using SIML, just as if either PICML or WSML were used independently. The whole is greater than the sum of its parts, however, because SIML defines new interactions that allow connecting a CCM component (or assembly) with a Web Service and automates generation of necessary gateways, which are capabilities that exist in neither PICML nor WSML.

B. Resolving Functional Integration Challenges using SIML

We now show how we applied SIML to resolve the functional integration challenges discussed in Section II-B in the context of our case study example described in Section II. Although we focus on the initial version of SIML that supports integration of CCM and Web Services, its design is sufficiently general that it can be applied to integrate many other middleware technologies without undue effort. Figure 5 shows how SIML resolves the following challenges to generate a gateway given an existing CCM application:

Resolving challenge 1. Choosing an appropriate level of integration. To allow interactions between CCM components and Web Services, SIML defines interactions between ports of CCM components and ports exposed by the Web Services. SIML thus extends the list of valid interactions of both CCM components and Web Services, which is an example of a composite DSML defining interactions that does not exist in its sub-DSMLs. SIML can also partition a large system into hierarchies via the concept of "modules," which can be either CCM components (and assemblies of CCM components) or Web Services.

In our case study example, we use SIML to define interconnections between the CCM and Web Service logging capabilities by connecting the ports of the CCM Logging Component to the ports of the Logging Web Service. These connections automate a number of activities that arise during

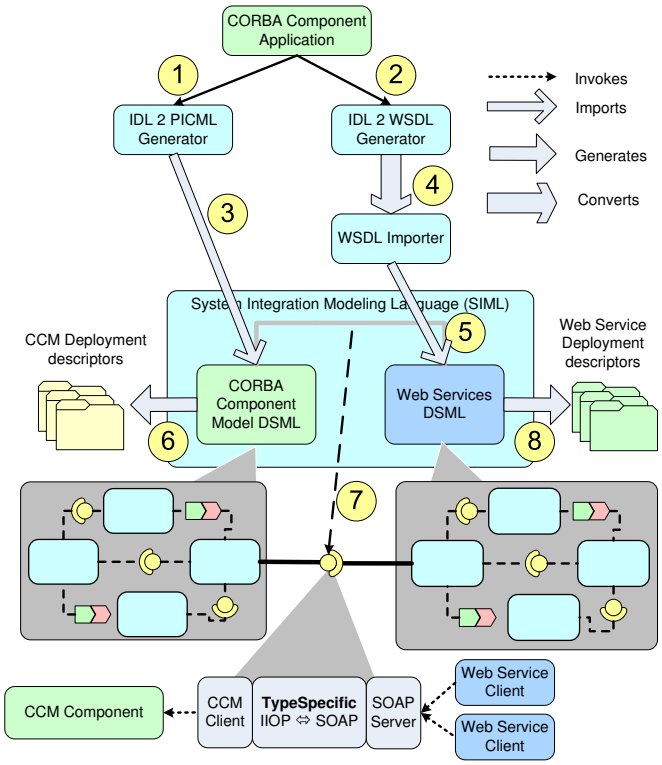

Fig. 5. **Generating a Web Service Gateway Using SIML**

integration including generation of resource adapters, such as the gateways shown in step 7 of Figure 5 and described in item 3 below. SIML therefore provides a ready-made framework for system integrators to define the points of interaction in their system, and avoids having to deal with low-level mechanics of the integration design.

SIML's architecture can be enhanced to support integration of many middleware technologies, by extending the list of interactions defined by SIML to integrate new technologies. For example, SIML could be extended to support interactions between CCM and EJB, or even between Web Services and EJB. The only requirement is to have a DSML that describes the elements and interactions of EJB.

Resolving challenge 2. Reconciling differences in interface specifications. To map interfaces between CCM and Web Services, SIML provides a tool called IDL2WSDL, which automatically converts any valid CORBA IDL file to a corresponding WSDL file. As part of this conversion process, IDL2WSDL performs both *datatype mapping*, which maps CORBA datatypes to WSDL datatypes, and *exception mapping*, which maps both CORBA exceptions to WSDL faults. System integrators are therefore relieved from the intricacies of the mapping. As shown in Figure 5, both IDL and WSDL can also be imported into the DSML environment corresponding to CCM (PICML) and Web Services (WSML), allowing integrators to define interactions between CCM components and Web Services. SIML also supports *language mapping* between ISO/ANSI C++ and Microsoft C++/CLI, which is the .NET framework extension to C++.

In our example scenario, IDL2WSDL can automatically generate the WSDL files of the Logging Web Service from the

IDL files of the Logging Component. The generated WSDL file can then be imported into SIML, and annotated with information used during deployment. SIML can also generate a WSDL file back from the model, so that WSDL stubs and skeletons can be generated. SIML therefore automates much of the tedious and error-prone details of the interface mapping, thereby allowing system integrators to focus largely on the business logic of the application being integrated.

Resolving challenge 3. Managing differences in implementation technologies. While the rules defined in SIML allow definition of interaction at the modeling level, this feature is not very useful if these definitions cannot be translated into runtime entities that actually perform the interactions. SIML therefore generates *resource adapters*, which automatically convert SOAP requests into IIOP requests, and vice-versa. A *resource adapter* in SIML is implemented as a gateway.

SIML allows system integrators to define connections between ports of a CCM component and a Web Service, as shown in Figure 5. These connections are then used by a *model interpreter*, which automatically determines the operation/method signatures of operations/methods of the ports on either end of a connection, and uses this information to automatically generate a gateway. The generated gateway contains all the "glue code" necessary to perform datatype mapping, exception mapping, and language mapping between CCM and Web Services.

The gateway generator is configurable and can currently generate Web Service gateways for two different implementation of Web Services: GSOAP [16] and Microsoft ASP.NET. The generated gateway also performs the necessary *protocol mapping* (*i.e.*, between IIOP and SOAP) and *discovery mapping* (*i.e.* automatically connecting to a Naming Service to obtain object references to CCM components). Our initial implementation does not yet support *QoS mapping*, which is the focus of future work, as described in Section VI.

In our case study example, SIML can automatically generate the Logging Web Service gateway conforming to GSOAP and/or Microsoft ASP.NET, by running the SIML model interpreter. Auto-generation of gateways eliminates the tedious and error-prone programming effort that would have otherwise been required to integrate CCM components with Web Services.

Resolving challenge 4. Managing deployment of subsystems. After the necessary integration gateways have been generated, system integrators also need to deploy and configure the application and the middleware using a variety of metadata in the form of XML descriptors. Since SIML is built using (meta)model composition it can automatically use the tools developed for the sub-DSMLs (*i.e.*, PICML to handle deployment of CCM applications and WSML to handle deployment of Web Services) directly from within SIML.

In our example scenario, SIML can thus be used to automatically generate the necessary deployment descriptors for all CCM components, as well as the Logging Web Service. SIML therefore relieves the system integrator from understanding low-level details of the formats of the different descriptors as well as the number of such descriptors required to deploy a CCM component or a Web Service.

By encapsulating the required resource adapters inside a a Web Service or CCM component, SIML allows reuse of deployment techniques available for any given middleware system. System integrators therefore do not need to deploy resource adapters separately. While this approach works for in-process resource adapters (such as those generated by SIML), out-of-process resource adapters need support from a deployment descriptor generator. Since SIML is a DSML itself, this support could be added to SIML so it can generate deployment support for out-of-process resource adapters.

Resolving challenge 5. Dealing with interoperability issues. Since knowledge of the underlying middleware technologies is built into SIML, it can automatically compensate for incompatibilities during design time. For example, IDL2WSDL allows generation of WSDL that supports SOAP RPC encoding or an interoperable subset defined in the WS-I Basic Profile. System integrators therefore are better prepared to handle incompatibilities that only show up during integration testing. SIML can also define constraints on WSDL definition as prescribed by the WS-I Basic Profile, so that violations can also be checked at modeling time. Similarly, gateway generation can add workarounds for quirks of particular implementations automatically, thereby relieving system integrators from finding these problems during final integration testing.

In our case study example, SIML can generate a Logging Web Service gateway that either supports a WS-I subset or uses SOAP RPC encoding. The DSML composition-based approach to integrating systems therefore relieves system integrators from developing more code during integration. The automation of gateway generation also scales the integration activity since developers need not write system specific integration code. In addition, SIML allows evolution of the integrated system by incrementally adding more components, or targeting different middleware implementations as future needs dictate.

C. Evaluating SIML

To evaluate the benefits of SIML, we first define a taxonomy for evaluating technologies that assist the functional integration of CCM and Web Services. We then use this taxonomy to compare SIML with tools that are supplied by vendors for either technology, referred to in Table I as *Native tools*. Examples of native tools include the Microsoft Visual Studio and the IBM Eclipse suite, which developers using middleware technologies like .NET and EJB are likely to use. This table depicts the different mapping activities described in Section IV-B that are typical in functional integration of middleware systems. For each activity the table describes the level of support in SIML and whether the activity is automated. It also describes the level of automation measured as the number of distinct steps performed by a system integrator.

Table I further decomposes the level of automation into three broad categories: (1) *design*, which denotes that system integrators need to perform a design activity that might include domain analysis, requirement analysis, etc., (2) *implementation*, which denotes that system integrators need to implement some functionality usually by writing code, and (3) *tool use*,

TABLE I
Evaluating Functional Integration using SIML

Integration Activity	Supported?	Automated?	Level of Automation (# of distinct steps)					
			Using SIML			Using Native Tools		
			Design	Implementation	Tool Use	Design	Implementation	Tool Use
Integration Design	Yes	No	0	0	1	1	1	0
Interface Mapping								
DataType Mapping	Yes	Yes	0	0	1	1	1	0
Exception Mapping	Yes	Yes	0	0	1	1	1	0
Language Mapping	Yes	Yes	0	0	1	1	1	0
Technology Mapping								
Protocol Mapping	Yes	Yes	0	0	1	1	1	0
Discovery Mapping	Yes	Yes	0	0	1	1	1	0
QoS Mapping	No	No	1	1	0	1	1	0
Deployment Mapping								
Descriptor Generation	Yes	Yes	0	0	1	0	0	1
Gateway Placement	No	No	1	1	0	1	1	0
Interoperability Mapping	Yes	Yes	0	0	1	0	1	0

which denotes that a tool needs to be used by the system integrators to perform that activity. This categorization assigns a weight commensurate to the skills of the individual responsible for carrying out the task in a particular organization.

Our taxonomy also assumes that design and implementation are orders of magnitude more difficult/time-consuming than tool use. In Table I, therefore, multiple activities of the same category are considered equal (and the table only uses 1 or 0), since the magnitude difference will likely dwarf any small number of steps of any particular category. To estimate the amount of effort required, we sum up each of the three columns (*i.e.*, design, implementation, and tool) and then multiply the result by the weight assigned to each category. For example, a reasonable assignment of weight for these activities might be 10, 5 and 1, for each of design, implementation and tool use. With this assignment, we can see that using SIML requires $2 \times 10 + 2 \times 5 + 8 \times 1 = 38$ distinct steps to achieve functional integration. In comparison, using just the native tools would result in $8 \times 10 + 9 \times 5 + 1 \times 1 = 126$ distinct steps to achieve the same. It should be noted that the number of steps will get reduced drastically as (and when) native tools add support for integration activities.

The numbers in Table I are for each unique unit of work per unique pair of source and target technologies, *i.e.*, for a single datatype mapping, a single exception mapping, a single protocol mapping. To calculate the total cost of integration, we must take into account both the number of distinct types/exceptions/languages, and the number of unique pairs of technologies being integrated.

Table I shows that SIML helps reduce the effort by reducing the design and/or implementation activities associated with integration to ordinary tool usage activities. For example, SIML effectively reduces the design and implementation effort required to perform the datatype, exception and language mapping, to a single step of tool use. This table also shows that similar gains can be achieved for complex tasks, such as protocol mapping (conversion between IIOP and SOAP in this case) and discovery mapping (conversion between CORBA Object References and Web Service URIs). Finally, the table reveals current gaps in our toolchain, *i.e.*, SIML does not perform QoS mapping or help with placement of resource adapters (or gateways), which remains as future work.

V. Related Work

This section surveys the technologies that provide the context of our work on system integration in the domain of large-scale distributed enterprise systems. We classify techniques and tools in the integration space according to the role played by the technique/tool in system integration.

Integration evaluation tools enable system integrators to specify the systems/technologies being integrated and evaluate the integration strategy and tools used to achieve integration. For example, IBM's WebSphere [17] supports modeling of integration activities and runs simulations of the data that is exchanged between the different participants to help predict the effects of the integration. System execution modeling [18] tools help developers conduct "what if" experiments to discover, measure, and rectify performance problems early in the lifecycle (*e.g.*, in the architecture and design phases), as opposed to the integration phase. While these tools help identify potential integration problems and evaluate the overall integration strategy, they do not replace the actual task of integration itself since these tools use simulation-/emulation-based abstractions of the actual systems. SIML's role is complementary to these tools: once the integration evaluation has been done using these tools, SIML can be used to design the integration, as well as generating the various artifacts required for integration.

Integration design tools. OMG's UML profile for Enterprise Application Integration (EAI) [19] defines a Meta Object Facility (MOF) [20] based metamodel for collaboration modeling, as well as activity modeling. MOF provides facilities for modeling the integration architecture focusing on connectivity, composition and behavior. The EAI UML profile also defines a MOF-based standardized data format to be used by the different systems to exchange data during integration, which is achieved by defining an EAI application metamodel that handles interfaces and metamodels for programming languages such as C, C++, PL/I and COBOL, to aid the automation of transformation. While standardizing on MOF is a step in the right direction, the lack of widespread support for MOF by various tools, and the differences between versions of XML

Metadata Interchange (XMI) support in tools lead to problems in practice. The primary difference between SIML and these tools is that SIML not only allows such integration design, but also automates the generation of key integration artifacts, such as gateways.

Integration patterns [21] provides guidance to system integrators in the form of best patterns and practices with examples of using a particular vendor's products. [4] catalogs common integration patterns with an emphasis on system integration via Message-Oriented Middleware (MOM) using different commercial products. These efforts do not directly provide tools for integration, but instead provide critical guidance to using existing tools to achieve integration. We are enhancing SIML to support modeling integration patterns and using them to enhance the generative capabilities of SIML to enable widely-accepted solutions to common integration problems.

Resource adapters are used during integration to transform data and services exposed by service producers to a form that is amenable to service consumers. Examples include data transformation (mapping from one schema to another), protocol transformation (mapping from one network protocol to another), or adaptation of interfaces (which includes both data and protocol transformation). While existing standards (such as the Java Messaging Specification [22] and J2EE Connector Architecture Specification [23]) and tools (such as IBM's MQSeries [24]) provide the architectural framework for performing the required adaptations, these tools approach the integration from a middleware and programming perspective, *i.e.*, system integrators are still required to handcraft the "glue" code that invokes the resource adapter frameworks to connect system components together. In contrast, SIML uses syntactic information present in the DSMLs to automatically perform the required mapping/adaptation by generating the necessary "glue" code, and relies on user input only for tool use.

Integration frameworks. Composition in the context of the semantic web and the Web Ontology Language (OWL) [25] has focused on composition of services from unambiguous, formal descriptions of capabilities as exposed by services on the web. Research on service composition has focused on automation and dynamism [26], optimizing the composition such that it is QoS-aware [27], as well as integration on large-scale "system-of-systems" like the GRID [28]. Since these automated composition techniques rely on unambiguous, formal representations of capabilities, system integrators need to make their legacy systems available as Web Services or provide alternate formal mappings of capabilities of the system to be integrated, which may not always be feasible. Our approach to (meta)model composition, however, is not restricted to a single domain, though the semantics are bound at design time. While both approaches rely on metadata, SIML's use of metadata focuses on the generative capabilities possible rather than on the semantic knowledge extracted from metadata.

Integration quality analysis. As the integration process evolves, it is necessary to validate whether the results are satisfactory from functional and QoS perspectives. Research on QoS issues associated with integration has yielded languages and infrastructure for evaluating Service-Level Agreements (SLAs). Examples include the Web Service Level Agreement language (WSLA) [29] framework, which defines an architecture to define service-level agreements using an XML Schema, and provides associated infrastructure to monitor the conformance of the running system to the desired SLA. Other efforts have focused on defining processes for distributed continuous quality assurance [30] of integrated systems to identify the impact on performance during system evolution. Information from these analysis tools should be incorporated into future integration activities. While these tools can be used to provide input to design-time integration activities, they themselves do not support automated feedback loops. We are adding support for modeling SLAs in SIML to allow evaluation of SLAs before/after integration.

VI. Concluding Remarks

The development of enterprise distributed systems increasingly involves more integration of existing COTS software and less in-house development from scratch. With the increase in capabilities of COTS component middleware technologies, the complexity of integration of systems built upon such frameworks is also increasing. This paper shows how a model-driven approach to functional integration of component middleware technologies enhances conventional approaches to system integration, which are tedious, error-prone, and non-scalable for enterprise distributed systems. We then show how DSMLs and (meta)model composition can help to address these limitations.

To demonstrate the viability of our approach, we developed the *System Integration Modeling Language* (SIML), which is a DSML composed from two other DSMLs, the CCM profile of Platform-Independent Component Modeling Language (PICML) and the Web Services Modeling Language (WSML). Finally, we evaluated the benefits of our approach by generating a gateway from the model, which automates key steps needed to functionally integrate CCM components with Web Services.

The following is a summary of lessons learned thus far from our work applying (meta)model composition to integrate heterogeneous middleware technologies:

- **Integration tools are becoming as essential as design tools.** SIML is designed to bridge the gap between existing *component technologies* (in which the majority of software systems are built) and *integration middleware* (which facilitate the integration of such systems). SIML elevates the activity of integration to the same level as system design by providing tools which allow integration design of systems built using heterogeneous middleware technologies. Since SIML is a DSML, it can potentially be used as the infrastructure to define constraints on the actual integration process itself, thereby allowing evaluation of service-level agreements prior to the actual integration itself.
- **Automating key portions of the integration process is critical to building large-scale distributed systems.** Compared with conventional approaches, our model-driven approach to system integration automates key aspects of system integration, including gateway "glue code" generation, metadata management, and design-time support for expressing unique domain and/or implementation assumptions. It supports

seamless migration of existing investment in models and allows incremental integration of new systems. Moreover, our model-driven approach is general-purpose and can be applied to tool-chains other than GME, as well as help integrate systems other than CCM or Web Services.

• **QoS integration is a complex problem, and requires additional R&D advances.** Though SIML helped map functional aspects of a system from a source technology to a target technology, our work is not complete until the non-functional QoS-related aspects of a system also map seamlessly. For example, technologies like the Real-time CORBA Component Model (RT-CCM) [31] support many QoS-related features (such as thread pools, lanes, priority banded connections, and standard static/dynamic scheduling services) that allow system developers to configure the middleware to build systems with desired QoS features. When systems based on RT-CCM are integrated with other technologies, it is critical to automatically map the QoS-related features used by an application in the source technology to the set of QoS features available in the target technology. For example, a number of specifications have been released for Web Services that target QoS features, such as reliable messaging, security, and notification. The focus of our future efforts in integration involves extending SIML to automatically map QoS features from one technology to another using DSMLs, such that the integration is automated in all aspects – both functional and non-functional.

REFERENCES

[1] Sun Microsystems, "Enterprise JavaBeans Specification." java.sun.com/products/ejb/docs.html, Aug. 2001.

[2] Object Management Group, *CORBA Components*, OMG Document formal/2002-06-65 ed., June 2002.

[3] Microsoft Corporation, "Microsoft .NET Development." msdn.microsoft.com/net/, 2002.

[4] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, October 2003.

[5] C. Britton and P. Bye, *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems*. Addison-Wesley Professional, May 2004.

[6] D. C. Schmidt, "Model-Driven Engineering," *IEEE Computer*, vol. 39, no. 2, pp. 41–47, 2006.

[7] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proceedings of the IEEE*, vol. 91, pp. 145–164, Jan. 2003.

[8] G. Karsai, S. Neema, B. Abbott, and D. Sharp, "A Modeling Language and Its Supporting Tools for Avionics Systems," in *Proceedings of 21st Digital Avionics Systems Conf.*, Aug. 2002.

[9] J. A. Stankovic, H. Wang, M. Humphrey, R. Zhu, R. Poornalingam, and C. Lu, "VEST: Virginia Embedded Systems Toolkit," in *Proceedings of the IEEE Real-time Embedded Systems Workshop*, (London, UK), IEEE, Dec. 2001.

[10] Ákos Lédeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti, "On Metamodel Composition," in *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, (Mexico City, Mexico), pp. 756–760, IEEE, 2001.

[11] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, pp. 44–51, November 2001.

[12] Anonymous, "Citation removed to assist blind review,"

[13] K. Ballinger, D. Ehnebuske, C. Ferris, M. Gudgin, C. K. Liu, M. Nottingham, and P. Yendluri, "WS-I Basic Profile." www.ws-i.org/Profiles/BasicProfile-1.1.html, April 2006.

[14] B. Meyer, "Applying Design By Contract," *Computer (IEEE)*, vol. 25, pp. 40–51, Oct. 1992.

[15] M. Emerson and J. Sztipanovits, "Techniques for metamodel composition," in *The 6th OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2006*, (Portland, OR), ACM, Oct 2006.

[16] R. van Engelen and K. Gallivan, "The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks," in *CCGRID*, pp. 128–135, IEEE Computer Society, 2002.

[17] IBM, "WebSphere." www.ibm.com/software/info1/websphere/index.jsp.

[18] C. Smith and L. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable*. Addison-Wesley Professional, Sept. 2001.

[19] Object Management Group, *UML Profile for Enterprise Application Integration (EAI)*, omg document formal/04-03-26 ed., March 2004.

[20] Object Management Group, *MetaObject Facility (MOF) 2.0 Core Specification*, OMG Document ptc/03-10-04 ed., Oct. 2003.

[21] D. TrowBridge, U. Roxburgh, G. Hohpe, D. Manolescu, and E. G. Nadhan, "Integration Patterns." msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/intpatt.asp, June 2004.

[22] SUN, "Java Messaging Service Specification." java.sun.com/products/jms/, 2002.

[23] S. Microsystems, "J2EE Connector Architecture Specification." java.sun.com/j2ee/connector/, November 2003.

[24] IBM, "MQSeries Family." www-4.ibm.com/software/ts/mqseries/, 1999.

[25] W. W. W. Consortium, "Web Ontology Language." www.w3.org/2004/OWL/, Feb 2004.

[26] S. R. Ponnekanti and A. Fox, "SWORD: A Developer Toolkit for Web Service Composition," Jan. 01 2002.

[27] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-Aware Middleware for Web Services Composition," *IEEE Trans. Softw. Eng.*, vol. 30, no. 5, pp. 311–327, 2004.

[28] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," *Computer*, vol. 35, no. 6, pp. 37–46, 2002.

[29] H. Ludwig, A. Keller, A. Dan, R. P. King, and R. Franck, "Web Service Level Agreement Language Specification." researchweb.watson.ibm.com/wsla/documents.html, January 2003.

[30] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan, "Skoll: Distributed Continuous Quality Assurance," in *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, (Edinburgh, Scotland), IEEE/ACM, May 2004.

[31] N. Wang and C. Gill, "Improving Real-time System Configuration via a QoS-aware CORBA Component Model," in *Hawaii International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, HICSS 2004*, (Kona, HW), HICSS, Jan. 2004.