Electronic Communications of the EASST
Volume 21 (2009)

Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009)

Synthesizing Executable Simulations from Structural Models of Component-Based Systems

Andreas Schuster and Jonathan Sprinkle

10 pages

Guest Editors: T. Levendovszky, L. Lengyel, G. Karsai, C. Hardebolle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Synthesizing Executable Simulations from Structural Models of Component-Based Systems

Andreas Schuster[1] **and Jonathan Sprinkle**[2]

[1] Technische Universität Berlin, andreas@email.arizona.edu
[2] University of Arizona, sprinkle@ECE.Arizona.Edu

Abstract: Experts in robotics systems have developed substantial software tools for simulation, execution, and hardware-in-the-loop testing. Unfortunately, many of these robotics-domain *software* infrastructures pose challenges for a robotics expert to use, unless that robotics expert is also familiar with middleware programming, and the integration of heterogeneous simulation tools. In this paper, we describe a novel modeling language designed to bridge these two domains in an intuitive visual representation. Using this metamodel-defined modeling language, we can design and build structural models of robotics systems, and synthesize experiments from these constructed models. The restrictions implicit (and explicit) in the visual language guide modelers to build *only* models that can be synthesized, a "correct by construction" approach. We discuss the impact of this language with a running example of an autonomous ground vehicle, and the hundreds of configuration parameters and several simulation tools that are necessary in order to simulate this complex example.

Keywords: Metamodeling, heterogeneous simulation, configuration synthesis

1 Introduction

The miniaturization of computing power has enabled bold advancements within the robotics community in recent years. Commodity hardware is now used for software-enabled sensing and control on vehicles that were previously limited to embedded control (i.e., micro-controller) technology. Chief among the technologies that permits control, computation, and communication on such robotic vehicles—and indeed, between teams of robotic vehicles—is middleware, which abstracts the location and operating system of various functional components.

Component-based software engineering (CBSE) is a system development approach that permits functional behaviors of components to be generalized for reuse. Components run asynchronously and exchange information through communication. In order to be generalized, many components can be developed with various parameters which are read at initialization time, and permit software reuse across various sensor families, actuator designs, or computational constraints and preferences [MBK07]. When used effectively, CBSE provides abstraction of communication (network, inter-process), distribution (multicore, distributed computers), implementation (language, OS), and interface (XML, objects, streams).

With these abstractions enabled, one of the major benefits of CBSE is the ability to partition the workload of system development into various chunks that can be independently developed,

validated, and subjected to regression tests on a regular basis [S+09]. For robotics systems, one difficulty in building reliable software arises from the task's complexity [MBK07] and its dynamic real-time nature. Additional factors, including the distributed and cross-platform computing environment, the large number of software contributors, and the desire to use existing code increase the difficulty. As projects increase in size, the complexity of running simulations or creating tests similarly increases. Many such projects employ domain experts who have developed a proof of concept of their algorithm in a CBSE language, incorporating the various middleware types into the specification of their algorithm.

In a robotic system such as an autonomous vehicle, algorithms for perception, navigation, control, and logging are mapped to a set of components. The integration of these components is generally completed using the mechanisms available in a middleware technology. However, some domain-specific abstraction can be additionally gained by the application of robotic-specific middleware, as seen in the projects Gearbox [MBK07], Player/Stage [CMG05], Orocos [BSK03], Microsoft Robotics Studio [Che07], etc. Such tools make it possible to define and develop building-blocks that can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks.

1.1 Configuration Explosion

Understanding the intersection of modeling and simulation (discussed in detail in [VLM02]) is a key goal of multi-paradigm modeling. In the application we discuss in this paper, the model of the system can help to manage the configuration explosion problem, while simultaneously serving as the necessary specification for simulation of the entire system.

Each component has a static configuration space, and in many cases, components can run with no configuration, based on value defaults encoded in the component software. However, in order to perform regression tests, or maintain the current running version of a robot, configuration files must be managed. From the perspective of good system design, ad hoc management has several problems. New members of a project may be encouraged to look at, or clone, existing configuration files to run the simulation. This is problematic if members do not understand the entire configuration space, or if configuration spaces of more than one tool overlap in a single file. For instance, information on default communication ports may be included in a server configuration file (an installation directory), and overridden in user files (a home directory), or even in simulation configuration files (the directory housing simulation-specific files).

However, the most frustrating part of configuration explosion is its impact on existing domain experts (not middleware experts), who cannot rapidly distinguish between errors in the code, and errors in configuration. Thus, tremendous effort could be wasted in determining whether component experts should be involved, or whether the system setup is using invalid assumptions or values from another computer or simulation scenario.

The final issue in configuration explosion is the ability to maintain a record of system configurations that produced desired, or undesired, results in a concise manner. Small modifications to a system's configuration, such as latency in a connection, which components run on which machine, etc., each require a completely new set of configuration options, and "rolling back" to those options either requires a disciplined policy of storage, or an integrated solution that permits regeneration of these options from a central location.

1.2 Structural Specification

In this paper, we propose an integrated modeling solution for component-based software systems which utilize layered middleware as the communication and implementation technology. Our provided solution is a constrained graphical modeling language that provides a semantic mapping to the configuration space of an existing robotics middleware package, namely the Orca Robotics platform. We provide the details of the structural specification of such experiments, and algorithms necessary to synthesize the configuration options for them. Further, we provide several informal design patterns that lend themselves to reduce copy/paste duplication.

2 Background and Related Work

Researchers have recently seen significant success in the application of component-based technologies to mobile robots. In this approach, large systems are decomposed into atomic components that can be reused in other applications. From a domain-specific perspective, each component provides some processed data or control inputs based on existing inputs, or samples from the physical system. Even if one applies this component-based approach, the application domain may be complex and characterized by large variability in hardware and software configurations. As a result, virtually every new system requires a custom design [MBK07]. Regrettably, software components may be difficult to understand, or lack documentation for their input/output relationships, and this may result in software being rewritten rather than reused.

In recent work a number of different projects for robotics and real-time systems have addressed the issue of code duplication through the creation of software frameworks [BSK03, S+02, MBK06, Che07] Common in each of these frameworks is the goal to improve software reuse: those that are component-based define how interfaces should be specified (for the development of new components), as well as how components can be assembled (for the application of existing components). Many of these projects provide a repository of already working components, thus enabling the rapid assembly of components with little or no *coding* effort. As we have pointed out, though, the effort to perform reconfiguration of such components can be nontrivial.

2.1 Existing CBSE Technologies

Middleware and component-based technologies facilitate the abstraction of communication, and location of computation. Microsoft's COM+, and .NET technology provide some degree of language freedom, though they require to some extent a Windows execution environment. Enterprise Java Beans, and SOAP, each support a similar technology, permitting an abstraction of provided data types for particular components.

CORBA is a powerful middleware implementation though it has high complexity for entry-level programmers (similar to those who might implement an expert's algorithm as a component). Nonetheless, once this barrier is passed, CORBA provides powerful abstractions and tools, as well as support for an impressive variety of operating systems. Another powerful middleware is the Internet Communication Engine (ICE) [Hen04], though it lacks the widespread support or acceptance that the CORBA standard achieves. Both CORBA and ICE utilize a generic interface definition language (IDL in CORBA, SLICE in ICE). The CORBA and ICE communities are

somewhat disjoint, though in this paper we do not comment on their relative advantages, leaving this as a design decision for implementors.

2.2 Distributed Real-Time Embedded (DRE) Systems

When it comes to distributed real-time and embedded (DRE) systems, applications have historically been manually programmed and customized [S+02]. To address the problems with developing and customizing DRE systems from scratch, there is growing interest in composing these types of systems using commercial off-the-shelf (COTS) hardware and software.

An existing approach to address middleware-specific component composition is the *Component Synthesis using Model Integrated Computing* (CoSMIC) project. The CoSMIC toolsuite is designed to (1) model and analyze DRE application functionality and QoS requirements and (2) synthesize CCM-specific deployment metadata for CIAO and QuO required to provision and enforce end-to-end QoS both statically and dynamically [S+02]. This toolsuite is similar to our approach, though we focus our work on families of experiments for collections of components for novice users, rather than advanced scheduling and deployment for generic CCM models.

2.3 Models of Heterogenous Software Simulation

There is existing work in the simulation of heterogeneous software systems. A large body of work for discrete event simulation, namely using the DEVS tool, provides significant insight into the technologies requires to simulate discrete event systems. The DEVS framework, or similar technology, can also be used as a uniting model of computation for systems using heterogeneous simulation environments.

The Command and Control (C2) Wind Tunnel is an example of heterogeneous simulation of software systems [G+09]. This application provides a unified model for the structure of the system specification, and uses code generation to provide language-specific hooks for a diverse suite of programming environments, including MATLAB, Java, C++, and Colored Petri Nets. Communication is performed through the HLA middleware, and networks are simulated at the packet level to permit sophisticated experimentation with latency and network attack.

2.4 The Orca Robotics Project

We describe in some detail the configuration space of the Orca Robotics Project, since this framework is the one to which we semantically map our structural models. Orca is an open-source software framework for developing robotic systems [MBK06]. It considers itself as an implementation framework and not an architecture, meaning that developers may run their systems on any kind of network as long as the interfaces of their components are appropriately connected. Operating systems known to support Orca components include Linux, QNX Neutrino, and Windows XP. Such a diverse array of operating systems permits hard, and soft, real-time, execution of algorithms, providing additional accuracy for sensed data if necessary.

Orca selected ICE as its middleware, and the structure and appearance of much of the Orca configuration space reflects design choices by ICE implementors, in order to take advantage of existing parsers and patterns in ICE. Our tool design goals include the ability to synthesize

configuration parameters for each component, as well as an executable version of the configured system. To concretely ground our synthesis approach, we give some abbreviated examples of configuration files, rather than specify their abstract syntax; however, the abstract syntax of these files was useful in the specification of the semantic mapping.

As a matter of interest, previous releases of the Orca framework provided a GUI to provide a similar method of specification of configuration files. However, the solution we discuss provides a more configurable, and also more structured, modeling environment, with significant opportunities for extension. Moreover, the later releases do not provide this GUI as most users of Orca are power users, and do not need the graphical syntax.

2.5 Contribution

We differ from the previous Orca GUI in the following ways: we provide an environment where previous experiments can be easily archived, and to which future experiments can refer (e.g., using the GME-specific types of reference, or types/instances). We differ significantly from the CoSMIC approach in that our goal is not to generically service middleware specification as an architecture-description language. Our approach is a more domain-specific one, for experts whose experiments are frequently repeated with various parameter values, or topologically changed with instances of existing/multiple components.

Thus, it is imperative that we permit synthesis of experiments from the topology of the component diagram, and various input/output relationships that exist by definition. Our approach provides a configurable graphical environment that will permit new users to easily configure their experiments and begin to use the framework with confidence.

2.6 Configuration Files

Configuration files list properties as key/value pairs, and they generally consist of four sections: (1) component identity; (2) provided interfaces; (3) required interfaces; and (4) component configuration options. Item (1) is the only required element of a configuration file, and it is needed to enable the communication between different components, as well as establish the unique identity of this component on the network. In Figure 1 we show an example configuration file.

Provided interfaces are specified by name, and this name is hashed together with the component identity and platform in order for others to have a unique subscription. Components that are sinks will not provide any interfaces. Required interfaces are specified similarly. One can either use “direct” or “indirect” binding, selecting either the interface name (and port ID), or the platform/component combination. Generally, an indirect binding is preferred, to permit the middleware to abstract the communication ports from the system constructor.

Configuration options can be grouped into set of arbitrary depth of the hierarchy, depending on the design choices of the component implementor. The `PARAM_TAG` is the name of the parameter being set. Generally, configuration options represent a static family of parameters with values that depend on the system under simulation or test. For example, configuration options may specify optimization choices of a controller, or the standard deviation of a sensor. For a particular piece of hardware, the configuration options are used to specify driver information and file information such as the serial port on which the hardware is located.

```
# Component
COMP_TAG.Platform=p1
COMP_TAG.Component=c1
COMP_TAG.Endpoints=e1
# Provided interfaces
COMP_TAG.Provides.IFACE_TAG1.Name=i1
COMP_TAG.Provides.IFACE_TAG2.Name=i2
# Required interfaces
# Direct binding
COMP_TAG.Requires.IFACE_TAG1.Proxy=i1:e1
# Indirect binding
COMP_TAG.Requires.IFACE_TAG2.Proxy=i2@p1/c1
# Configuration Options
COMP_TAG.Config.SET_TAG1.PARAM_TAG1=0.1
COMP_TAG.Config.SET_TAG1.SET_TAG2.PARAM_TAG2=1
```

Figure 1: Sample component identity, provided, and required sections.

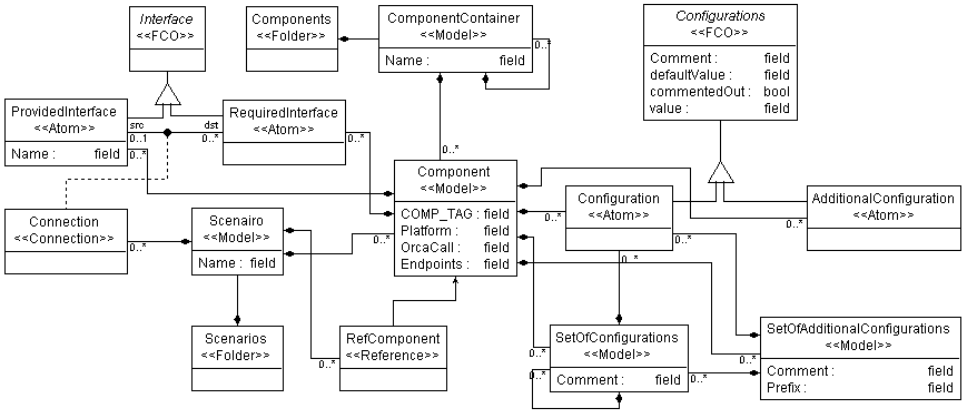

Figure 2: Metamodel

3 Language Specification

Our approach to provide an intuitive specification environment, as well as a fully-featured constraint-based language involves the development of a metamodel. We developed the language and all translators in the GME toolsuite [LBM+01].

Our language centers around the *Component* type. As one can see in figure Figure 2 it contains four attributes described in table Table 1. We discuss briefly the significance of the *InstanceName* attribute. In the event that a platform requires more than one instance of the same component (e.g., multiple laser scanners on a platform), then each component can use a different configuration file. Importantly, the application that simulates or interfaces with the laser scanner would be the same both times, but would read from a different configuration file.

Component objects represent interface and configuration options through containment. *ProvidedInterface* types correspond to interfaces to which another component can subscribe. Like-

Table 1: Attributes of the Type Component

GME Attr	Orca Configuration Mapping
COMP_TAG	Maps to the `COMP_TAG`
Platform	Maps to the key `Platform`
Endpoints	Maps to the value to the key `Endpoints`
InstanceName	Instance-specific name of this component and associated configuration file

wise, *RequiredInterface* types correspond to interfaces to which this component must subscribe. *Configuration* types and subtypes can be composed in order to permit the hierarchical structure shown in Figure 1. Using the various parameter values of these types, boolean, string, integer, and floating point parameters can be enforced for parameter types. The type *SetOfConfigurations* enables the modeler to group configurations together, where all configuration options in a set of configurations will have the same `SET_TAG`. In addition a set of configurations can contain other set of configurations. This allows the modeler to nest sets in a arbitrary depth of hierarchy.

Semantically *Configuration* and *AdditionalConfiguration* differ in that the `COMP_TAG` is not found for generated text of *AdditionalConfiguaration* options unless explicitly given as the value. This permits compatibility with components developed outside the style guide of Orca, and is extended for *SetOfAdditionalConfigurations* to permit composition.

In order to enforce the so called *static semantics* of our language we developed constraints, which restrict some (normally allowed) syntax patterns when used in certain contexts. These constraints are expressed in an extension of OCL available in GME, and can be seen in Figure 3. One constraint is that the attributes *COMP_TAG* and *InstanceName* are explicitly set. They are both absolutely essential, and this constraint ensures that they are not omitted accidentally[1]. The more subtle constraint attached to *RequiredInterface* types ensures that a required interface is connected to no more than one provided interface. While a provided interface can deliver data to any number of required interfaces, it does not make sense to have more than one source for one required interface.

After the modeler has built her model of this scenario, she wants to generate the output files that will represent and execute this experiment. The semantic map from the metamodel to our application domain performs this mapping through a series of traversals.

Our first traversal produces one configuration file for each component in this scenario, a straightforward process based on a direct map from the component model's contained configuration options, and the source and destination component ports for each produced and required interface. The second traversal examines the data dependencies of the component graph, and

[1] Note that in GME, graphical cardinalities at the diagram level are not enforced for associations.

```
Component.COMP_TAG.trim() <> ""
...
RequiredInterface.connectedFCOs("src")->size <= 1
```

Figure 3: Constraints

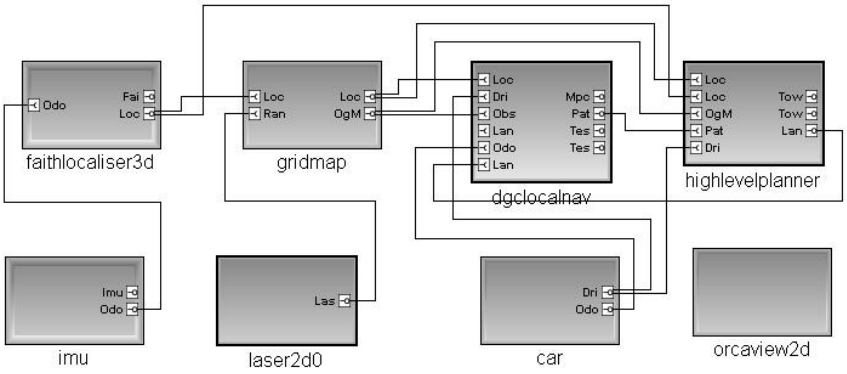

Figure 4: Running Example

synthesizes a standalone script to start up components in the appropriate order. Included in this script is the necessary order and timing of the startup of third party tools such as a physical system simulator, and middleware core services such as the registry and cache databases.

4 Example Simulation

Figure 4 shows an example model, which evolved during the development of our language. Each of the eight blocks in Figure 4 represents one component, and makes up the simulation of an autonomous vehicle. The model interpreter examines the data dependencies of the component models, and produces a script that (1) starts up necessary network interfaces, (2) initiates physical/network simulators, (3) initializes and starts each component with the specified parameters, (4) logs the data to a unique output file. This simulation capability is key to our approach, in that the dependable startup order and parameter values are explicit and implicit in our specification, rather than depending on user training.

We used three different runtime types to put together this model. The blocks with the thickest frames (`faithlocaliser3d` and `imu`) are models which were created inside the `Scenario` model. The blocks with the same type of, but thinner frame (`gridmap`, `car` and `orcaview2d`) are instances of prototype models that are created within a component container; i.e., they are instantiations of a library of components. The third type of blocks with the solid, black frame (`dgclocalnav`, `highlevelplanner` and `laser2d0`) are references (pointers) to models that are created in a component container.

It is useful to have these alternatives for reuse purposes (e.g., a database of components). Such a database can be organized using component containers. Now she has two ways to put those components in a scenario: copy them as instances of library prototypes, or point components to these reference library components. If the modeler wants to have more than one experiment which include the same component with the exact same attributes, she will want to use references. Because references point to an original model, the attributes of that original model cannot be overridden by the pointer, and are guaranteed to propagate to all scenarios that point to them. This way a modeler can apply changes at a central point.

If the modeler wants to use the same component, but with different attributes, within the same experiment, she will want to use instances of prototypes, or copies of the models. Instances of prototype models have the same structure and attributes of those prototypes, but one is able to change the values for those attributes independently from each other. If a modeler wants to provide her component database to the public, but has a few components that are not supposed to be published, she will be able to create those components directly in an experiment.

5 Analysis

For the simulation presented in Section 4 a total of 360 lines of code were generated: 70 in the startup script and 290 in configuration files. Although a domain expert can create the script file rapidly (through cloning another one), this is not feasible for new users of the simulation environment, and thus the startup costs for a new user are significantly reduced through the use of this environment. Once a model in our language exists, it is a question of seconds to generate all files at once. The steps needed to uniquely reconfigure an experiment have also been reduced. Component addition or deletion is now handled graphically, and component interconnection is trivial with line-drawing capabilities (and explicit constraints) of the tool. Earlier when including a new component, one should be familiar with existing configuration files to reuse them as new. Now, it is enough to create a visual representation (model) of the new component and the configuration file will be generated automatically.

Another advantage of our visual language that should not be underrated is the documentation that comes with the creation of a model of an experiment. The model shows what components take part and how they are connected to each other. It now is very easy for an robotic expert and even for a layman to understand the overall design.

This perspective of our approach distinguishes this from an approach that does not involve model-based design. In fact, the number of users of the Orca environment is somewhat limited, due to the expertise needed to understand the experiment topologies well. We believe that our approach will permit a wider audience to take advantage of the Orca toolbox. Further, we can now take advantage of many of the benefits of the GME toolsuite, to rewrite component diagrams for advanced purposes, such as synthesis of time-triggered scheduling buffers [SE09].

Although our example is certainly possible to produce without a model-based approach, the model-based approach rapidly introduces new team members to the design, and permits them to make contributions much more quickly. Future work with this environment has the potential to reduce effort for even advanced users.

6 Conclusion and Future Work

The ability to rapidly synthesize robotics experiment configurations is a first step in an integrated modeling approach for this largely ad hoc community. An interesting next step is the introduction of domain-specific patterns in stubs or shell programs based on an existing set of required and provided interfaces. The amount of duplicated code between components to produce these shells is significant, and such stubs would dramatically improve the chances of a new user adopting a particular robotics middleware solution.

Bibliography

[BSK03] H. Bruyninckx, P. Soetens, B. Koninckx. The Real-Time Motion Control Core of the Orocos Project. In *IEEE International Conference on Robotics and Automation*. Pp. 2766–2771. 2003.

[Che07] S. Cherry. Robots Incorporated. *Spectrum, IEEE* 44(8):24–29, Aug. 2007.

[CMG05] T. Collett, B. MacDonald, B. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*. 2005.

[G+09] J. Gulotta et al. Using Integrative Models in an Advanced Heterogeneous System Simulation. In *Engineering of Computer-Based Systems, IEEE International Conference on the*. Pp. 3–10. IEEE Computer Society, Los Alamitos, CA, USA, 2009.

[Hen04] M. Henning. A new approach to object-oriented middleware. *IEEE Internet Computing* 8(1):66–75, 2004.

[LBM+01] Ákos Lédeczi, Árpad Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer* 34(11):44–51, November 2001.

[MBK06] A. Makarenko, A. Brooks, T. Kaupp. Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*. Pp. 163–168. 2006.

[MBK07] A. Makarenko, A. Brooks, T. Kaupp. On the Benefits of Making Robotic Software Frameworks Thin. In Prassler et al. (eds.), *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*. November 2007.

[S+02] D. Schmidt et al. CoSMIC: An MDA generative tool for distributed real-time and embedded component middleware and applications. In *OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture, Seattle, WA*. 2002.

[S+09] J. Sprinkle et al. Model-based design: a report from the trenches of the DARPA Urban Challenge. *Software and Systems Modeling* 8(4):551–566, September 2009.

[SE09] J. Sprinkle, B. Eames. Model-Based Autosynthesis of Time-Triggered Buffers for Event-Based Middleware Systems. In *9th OOPSLA Workshop on Domain-Specific Modeling*. P. (to appear). October 2009.

[VLM02] H. Vangheluwe, J. de Lara, P. Mosterman. An introduction to multi-paradigm modelling and simulation. In *Proceedings of the 2002 Conference on AI, Simulation, and Planning*. Pp. 9–20. 2002.