Invited Paper

On Syntactic Composability and Model Reuse *

Claudia Szabo, Yong Meng Teo

Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
E-mail: [claudias,teoym]@comp.nus.edu.sg

Abstract

Composability, the capability to select and assemble off-the-shelf model components in various combinations to satisfy user requirements, is an appealing approach in reducing the time and costs of developing complex simulation. This paper discusses CODES, a hierarchical component framework to support component-based modeling and simulation. We propose the use of EBNF based grammars to specify syntactic composability rules with the aims of achieving syntax consistency for model components to operate together. EBNF production strings associated with each composed models are transformed into a unique identifier to support distributed DHT-based model discovery. The hierarchical design supports the sharing and reuse of models and model components across application domains, and facilitates the verification of composed models. We present a prototype of the framework implemented using the Scalable Simulation Framework, and illustrate this approach by modeling a grid computing system.

1. Introduction

In modeling and simulation, there is a growing interest for developing larger and more complex models through model composition [5], [7], [14], [27]. The defining characteristic of composability is the flexibility to combine and recombine simulation components into different simulation systems to meet diverse needs [18]. In discussing composability, there are different views in distinguishing the kind of problems that arise. Tolk [25] proposes different levels of interoperability ranging from no interoperability to conceptual interoperability and introduces the concept of "pragmatics" to encapsulate simulation context. Petty and Weisel [19] define nine levels of composability from *applications* to *behavior* and propose two perspectives to study composability, engineering and modeling. Syntactic (engineering) composability determines whether the components can be connected. To be syntactically composable, simulation components have to be compatible with respect to data passing mechanisms and timing assumptions. In contrast, semantic (modeling) composability addresses whether the combined computation of the composed simulation system is semantically valid.

Our motivation is two folds. While the approach to simulation development has progressed from program-centric to model-centric, simulation model composability and reuse have proved much more difficult to realize than some initially imagined [2]. Recent interests in building component-based simulations have resulted in various approaches and theories [4], [6], [18], [13]. Some approaches target component-based simulations in general, but most focus on specific application domains. Examples of application domain based simulation frameworks include electronic systems [7], thermofluid systems [20], and network [21] simulations. While application domain based component frameworks are perceived to be easier to develop, approaches that generalize and share components across application domains will increase the level of component reusability, and facilitate the development of more complex simulations at lower costs. Next, there is an increasing trend in using the Internet as an infrastructure for more pervasive sharing of resources. By leveraging on the recent advances in Internet technologies such as peer-to-peer, grid computing and web services, composable modeling and simulation will advance knowledge sharing to a larger and wider simulation community. It will also offer a quantum leap in both the capability and scale of the applications of simulation.

This paper addresses *syntactic composability*. Chen and Szymanski [4] adopt a black box component view and use C++ templates to achieve syntactic composability. To

*Proceedings of the International Conference on Modeling and Simulation, pp. 230-237, IEEE Computer Society Press, Thailand, March 2007.

demonstrate interoperability in supply chains simulations, Verbraeck [27] utilizes a building block paradigm. A Fractal component model, proposed by Dalle [6], employs the notion of shared components to facilitate the use of the same common component across different components that encapsulate it. By far easier to achieve than semantic composability [1], syntactic composability still poses a number of problems such as establishing a common component model by which all components involved in the syntactic composition must abide, and, derived from the component model, the way in which syntactic checking is done. Another problem derives from the heterogeneity of simulation components and application domains in which they are used, since many component oriented simulation frameworks developed today are application domain oriented. This approach achieves greater depth in the coverage of a particular application domain, but lacks the scalability and complexity of simulation models offered by generalized component oriented frameworks. A main challenge is to achieve both *breadth* (many domains) and *depth* (detailed specific domains) coverage, and at the same time allow for facile cross-domain component integration, verification of assumptions and constraints [23]. CODES (COmposable Discrete-Event scalable Simulation) attempts to address these issues. The main contributions of this paper are: a hierarchical framework for sharing and reusing model components across application domains, an EBNF-based grammar for specifying composition and reuse rules that simplifies syntactic composability verification, and the exploitation of grammar production strings in the discovery of shared components for reuse.

This paper is organized as follows. Section 2 discusses the design of CODES covering its hierarchical architecture, composition and reuse using EBNF grammar and an example of a grid computing system simulation model. In Section 3, we discuss a prototype of CODES implemented using the Scalable Simulation Framework. A summary and discussion of future work are presented in Section 4.

2. Framework Design

Four steps can be distinguished when building component based simulations: component discovery, model validation, model execution and model deployment. Figure 1 shows the CODES framework that supports these steps. The model composer module is responsible for component discovery and for model validation. A model composer, a human or a software system, composes models from other models or model components. A *model repository* is a database for models or model components from which one may compose other models. A model composer needs the ability to process the intention of model composition, the ability to formulate a set of search criteria, access a model repository of properly annotated (or indexed) models and model components, perform relevant assessment of plausible models and model components, and compose a model from selected other models and/or model components into a syntactically and semantically coherent simulation system that satisfies user requirements. Given a conceptual model, plausible components for composition are discovered and retrieved from a model repository. To support scalable and efficient model discovery, model repository can be geographically distributed with a model indexing scheme [17]. A composed model undergoes syntactic and semantic checks by the *Validator* module before the validated simulator is passed to the actuator module which performs the model execution. The *Actuator* module supervises component interaction, time management, data management and result reporting. The *Distributor* module places the validated simulators in the model repository according to the deployment scheme to facilitate model discovery. A CODES GUI provides a graphical interface to support model development and execution.

To achieve generality across application domains and to support application domain based requirements, a hierarchical component architecture is proposed in Figure 2. CODES component repository consists of shared and reusable components common to all application domains, and shared components defined in each application domain. An application domain (AD) defines its pool of *base components* specific to the application. Composed models in an application domain are in turn placed in the CODES repository for sharing and reuse.

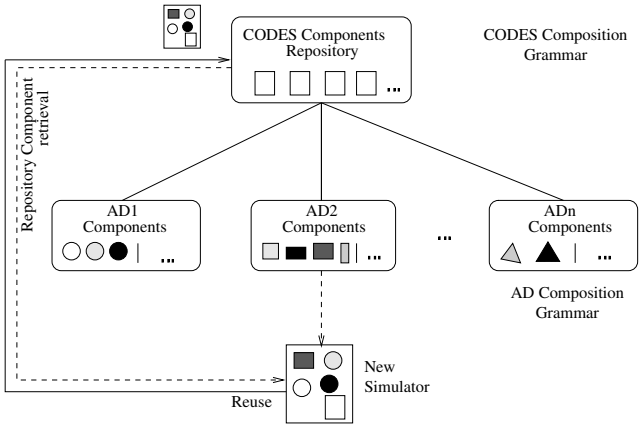

Figure 2. CODES Hierarchical Component Architecture

A CODES simulator adheres to our proposed *component-connector paradigm* and allows for syntactic composability and syntax checking during model development. In the component-connector paradigm, a *component* is viewed as a black box with an in- and/or

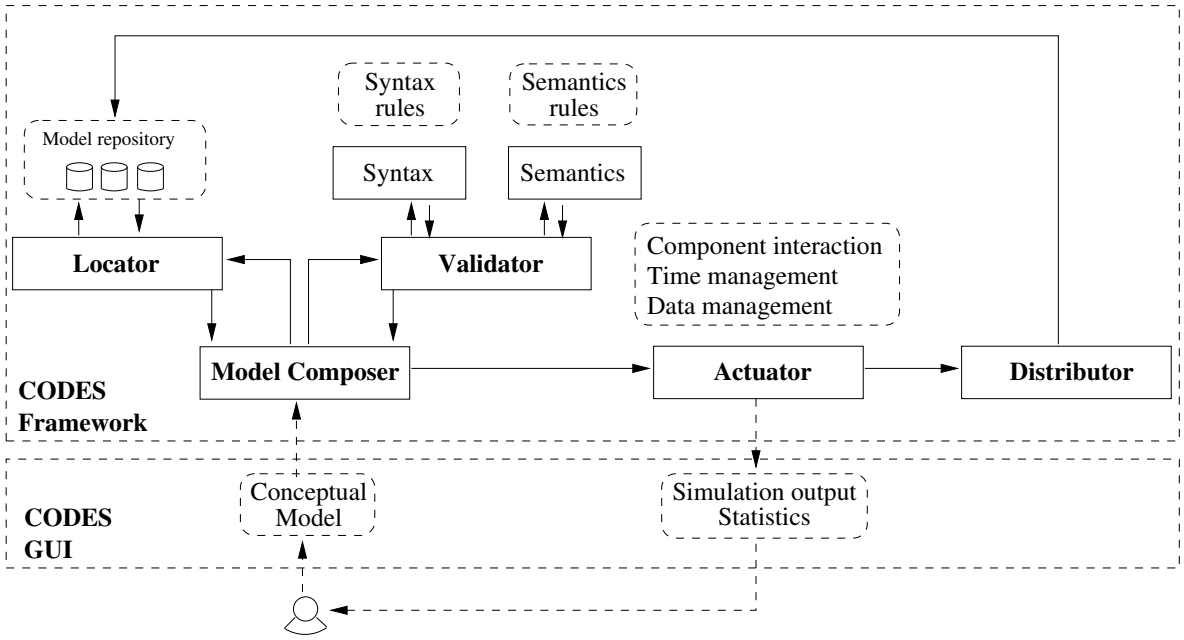

Figure 1. Component-based Simulation Model Development

an out-channel. Components are interconnected by *connectors*. A *connector* performs message and data passing among components. Messages leaving components are timestamped and the connector guarantees FIFO delivery of messages to the destination components. Connectors are divided into *one-to-one* for connecting two components, *many-to-one* for joining out-channels of components into one in-channel of the next component, and *one-to-many* for demultiplexing the out-channel of a component into in-channels of more than one component. The component-connector paradigm allows for loose coupling [22] of components and can be exploited to support component-oriented distributed simulations. To formalize the component-connector paradigm, composition grammar is used to define general model composition rules, and application domain specific composition rules are defined by application specific composition grammar.

While a CODES component is viewed as a black-box, a number of default and extensible attributes and methods are defined to support component-based model development. A component specification includes a user-defined *name* and a component *type* to differentiate type of base component. Component *attributes* define component characteristics such as the interarrival time of jobs, number of jobs in the system, queueing policy, and default performance metrics such as server utilization and average queue length. Additional attributes can be defined by extending the component, using the component's extension API. To achieve and verify syntactic composability, only the component name and type are used. *Constraints* [23] and formalized *behavior* such as state graphs are used to validate semantic composability. This is beyond the scope of this paper.

2.1. Syntactic Composability and Reuse

To support syntactic verification of composed models and to formulate search criteria to discover plausible models and model components, we propose to specify model composition and reuse rules using regular grammars. This approach not only has the advantage of formalizing syntactic composability but also provides a method of representing composed models as production strings, based on the grammar production rules fired. This provides a simple and efficient method of searching plausible models in the repository to support reuse. Syntax checking is performed by the Validator module based on the CODES composition grammar and application domain grammars. Figure 3 shows the CODES composition and reuse grammar in EBNF [9]. We illustrate application domain grammars using the example of queueing networks simulation. The composition grammar formalizes model composition and reuse rules in the component-connector paradigm. Application domain grammars define specific composition rules using application domain base components.

A CODES simulator consists of a set of model components *(Comp)* interconnected by connectors *(Con)*. Comp is a base CODES component *(B_Comp)* selected from an application domain such as queueing networks *(QN_B_Comp)*, or a reused component *(Rep_Comp)*, from the shared CODES component repository. Components are intercon-

```
# CODES Composition Rules
Simulator       ::= (Comp Con)+
Con             ::= ConO | ConF | ConJ
Comp            ::= B_Comp | Rep_Comp
B_Comp          ::= QN_B_Comp

# CODES Reuse Rule
Rep_Comp        ::= QN_R_Simulator | QN_E_Simulator

# Application Specific – Queueing Networks (QN)
# Base Components
QN_B_Comp       ::= Source | Server | Sink

# QN Composition Rules
QN_Simulator    ::= Source BlockNT+ Terminal?
                  | Source BlockT+
Terminal        ::= ConO Final | ConF ("(" Final ")")+
                  | ConJ Final
Final           ::= Source | Sink
BlockNT         ::= ConF ("(" BlackBox BlockNT*
                  (ConJ BlackBox BlockNT*)?")")+
                  | (ConO BlackBox BlockNT?)+ | _
BlockT          ::= ConF ("(" BlackBox BlockT* (Terminal |
                   ConJ BlackBox BlockT*)?")")+
                  | (ConO BlackBox BlockT?)+ | _
BlackBox        ::= Server | Rep_Comp
```

Figure 3. CODES Composition and Reuse Grammar

nected by connectors *(Con)* such as one-to-one *(ConO)*, fork or one-to-many *(ConF)* and join or many-to-one *(ConJ)*. As shown in Figure 4, base components in a queueing network application include source, server and sink. Domain specific composition rules define the connectivity of the basic components to form different queueing network systems. To allow for component reuse, a queueing network simulator can include base component such as a server *(Server)* or model components from the repository *(Rep_Comp)*. The composition rules allow for the composition of a plethora of open, closed and hybrid queueing networks [3]. The composed simulator is in turn placed in Rep_Comp as a model component for reuse.

2.2. Example: Simulation of a Grid Computing System

Simple examples of open, closed, and hybrid queueing models are discussed in [24]. In this section, we discuss the modeling of a grid computing system. Firstly, we model this open system using base queueing networks components as shown in Figure 4. Next, we show how the same system can be developed by reusing model components from the model repository. *Source* is a base queueing network component but its behavior differs depending on whether the system is an open or a closed queueing network. For open queueing systems, a source waits for a sampled interval of time, generates a job, and passes the job through its connector to the next component. In a closed system, a source waits for the completion of a job it releases into the system before putting the next job into the system. A *server* component encapsulates one or more service units and is served by a single queue. Jobs completed in an open system terminate at the *sink*.

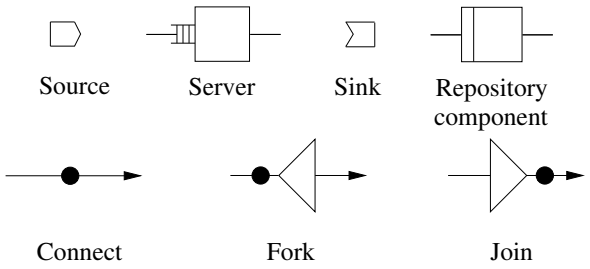

Figure 4. CODES Component and Connector Icons

Figure 5 shows a conceptual model of a simple grid system [11] with two virtual organizations (VO) [11] sharing a grid meta-scheduler job queue [26]. Each virtual organization consists of a local job scheduler and different types of computational resources (C).

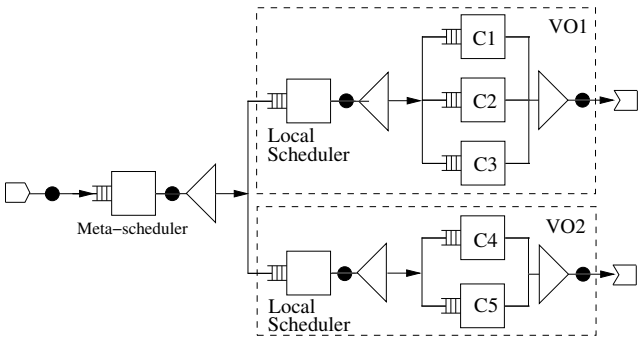

Figure 5. A Grid Computing System using Base Components

The production string for this model is

Grid_QN= Source ConO Server ConF
(Server ConF (Server ConJ Sink)
(Server ConJ Sink)
(Server ConJ Sink))
(Server ConF (Server ConJ Sink)(Server ConJ Sink))

A composed model can be reused as it is, i.e., *a model*, by storing the model in the repository with its production string to support model discovery.

A model to be reused as part of another larger simulation model is called a *model component*. For example, VO1 and VO2 in Figure 5 could have been developed earlier and deposited into the model depository as model components. A CODES model component must contain only one in- and one out-channel and is deemed *Rep_Comp* in the composition grammar.

The production strings corresponding to model components, VO1 and VO2, are respectively

VO1_QN = Server ConF (Server ConJ)
(Server ConJ)
(Server ConJ)
VO2_QN = Server ConF (Server ConJ) (Server ConJ)

Figure 6 shows the same grid system composed using model components, VO1_QN and VO2_QN, from the model repository. The corresponding production string for this reused model is

*Grid_QN= Source ConO Server ConF (**Rep_Comp** Sink)*
*(**Rep_Comp** Sink)*

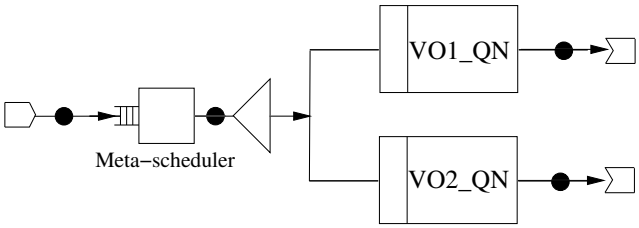

Figure 6. Grid System Composed by Reusing Model Components

3. Implementation using SSF

A prototype of CODES has been implemented using the Scalable Simulation Framework (SSF) [5]. SSF provides a Java API for building discrete-event simulations using a process-oriented modeling worldview. The concept of a SSF *entity*, a container of system state with "in" and "out" communication channels that are monitored by processes associated to the Entity, is synonymous to CODES component. Each process associated to an entity performs various computations and can change the attributes of an entity. Communication between entities is done through the use of event objects that flow through the communication channels.

Figure 7 presents the class diagram for the queueing network components. C_Source, C_Server and C_Sink correspond with the queueing network base components source, server and sink respectively. CODES components and connectors are implemented as SSF entities with their respective processes.

Each component consist of an *in* and *out* channel, as well as a *connector* field that points to the connector linking its out-channel. Different types of connectors are generalized using a generic SSF connector entity with left and right attributes defining the number of in- and out- links. Default metrics provided include server utilization and average queue length and are included in the ServerMetrics class. A component can be extended by adding new attributes or metrics. An extended component inherits the base component type, but changes its name, thus ensuring backward compatibility with respect to syntactic checking. Component extension is implemented using Java inheritance mechanisms.

The conceptual model specified in a configuration file is used by the ComponentManager class to build the composed model and verify the composition syntactically. The syntax check is done by calling the SyntaxChecker module with the configuration and grammar files. The SyntaxChecker module then parses the configuration file and determines the string that describes the composition which is then checked against the grammar using a grammar acceptor built based on Earley's algorithm [8]. For it to be parsable using the grammar acceptor, the CODES Composition Grammar and any subsequent application domain composition grammar need to be regular unambiguous grammars [12] and be expressed in EBNF [9] notation. For the queueing networks application domain the grammar used by the acceptor is represented in an easy to parse way, determined through the use of a finite non-deterministic automaton.

For ease of composable model development, we are currently developing a graphical model composition environment. A model composer begins by composing the conceptual model using base components. Discovery of shared model components is achieved by highlighting (selecting) a subset of components that prompts the GUI to generate a search query to lookup the model repository with the associated model production string. Figure 8 shows a snapshot of the GUI for the example grid system discussed.

4. Conclusions

The presented CODES framework offers an integrated approach in achieving both breadth and depth in the sharing and reuse of model components across application domains. In contrast to current approaches, we exploit EBNF based grammars to specify model composition rules so as

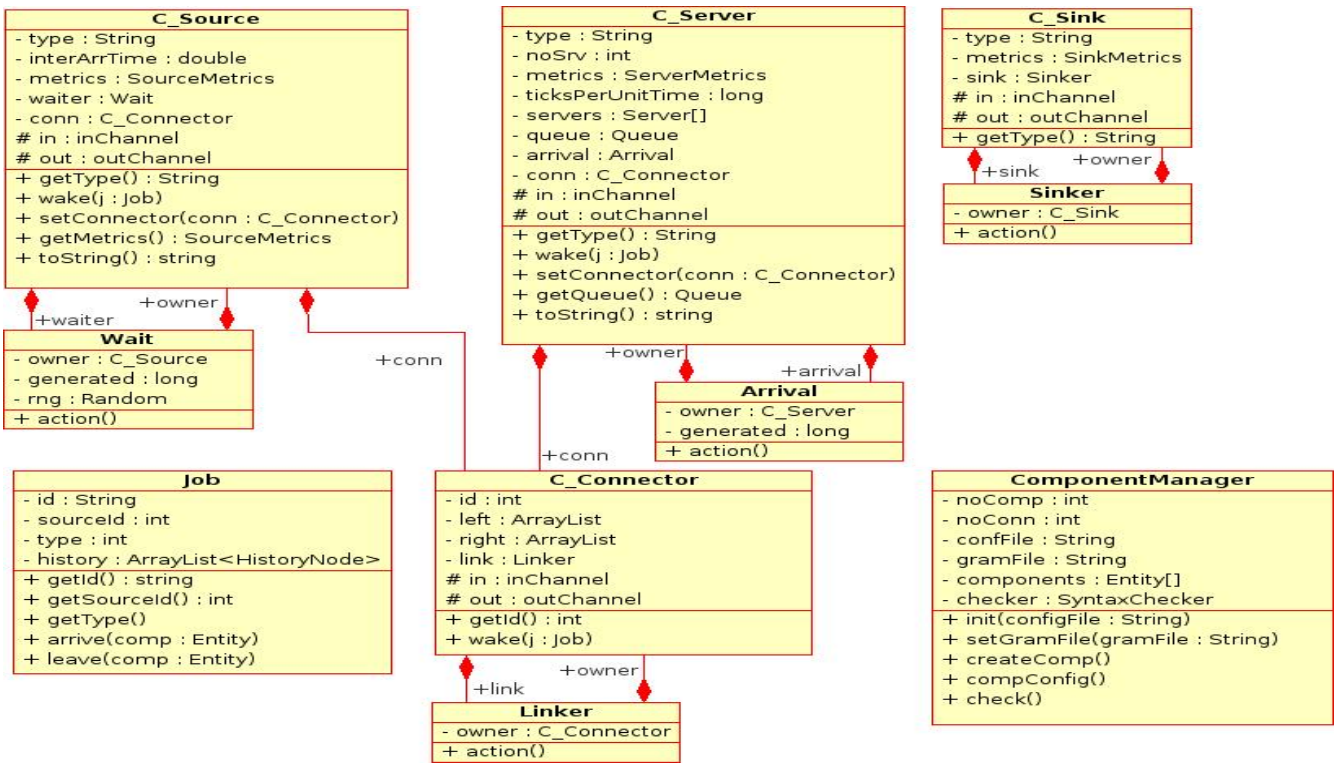

Figure 7. SSF Implementation of Base Components

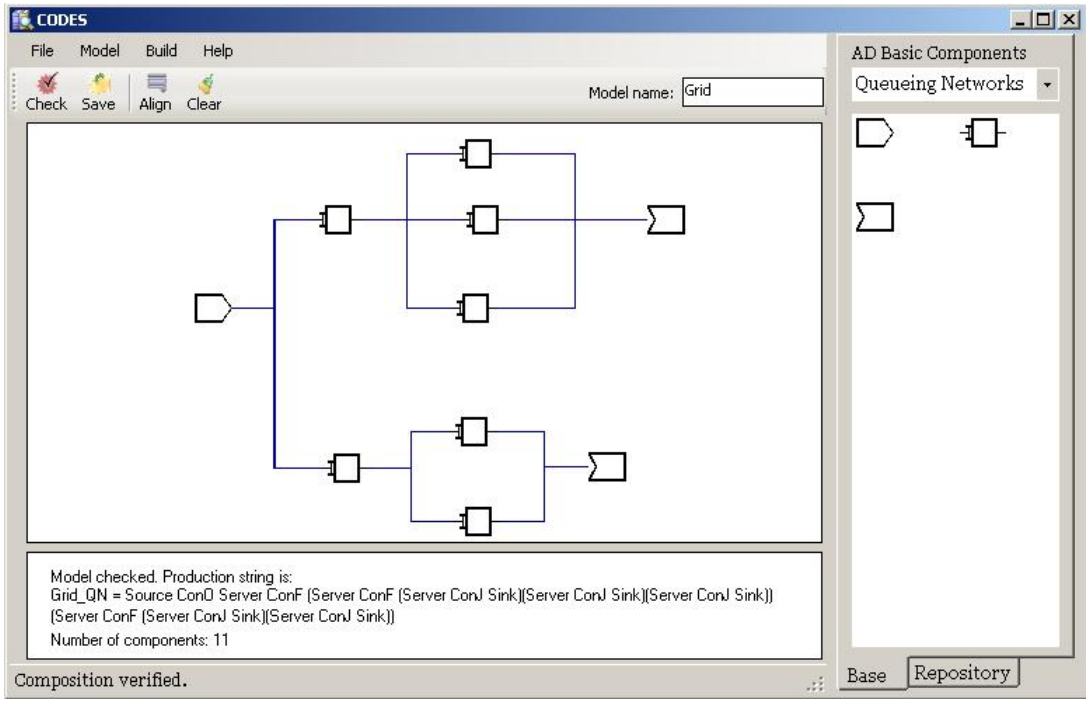

Figure 8. CODES Model Composition GUI

to support syntactic composability verification and discovery of shared models and model components. This provides a more structured approach to syntactic verification of composed models and also offers extendability to include new application domains. We are currently developing methods and techniques to model and represent component semantic. Semantic composability encapsulates validation of the behavior of components as a whole, and with respect to specific requirements and the constraints and attributes of components. This leads to various approaches. Petty and Weisel [18] propose a formal theory of composability in which a model is viewed as a computable function and the composition as a set of composed functions, leading to a complex mathematical apparatus for proving semantic validity. Miller and Fishwick [15] investigate the use of ontologies for simulation modeling and propose the Model-Component class to describe models using their composite building blocks. Moradi et al. [16] investigate the Base Object Model which offers an implementation independent view on a simulation component. Semantic composability is a bigger challenge and is an active area of research in which there is a lack of a clear consensus towards semantic validity. We are exploring the use of a component formalism such as DEVS [28] to describe CODES components, which could be in turn used by an Architecture Description Language [10] to formally specify and validate the semantics of composed models.

References

[1] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. In search of the philosopher's stone: Simulation composability versus component-based software design. *Proceedings of the Fall Simulation Interoperability Workshop*, 2004.

[2] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. The computational complexity of component selection in simulation reuse. *Proceedings of Winter Simulation Conference*, pages 2472–2481, 2005.

[3] S. Bose. *An Introduction to Queueing Systems*. Kluwer Academic/Plenum Publishers, 2002.

[4] G. Chen and B. Szymanski. COST: A Component Oriented Discrete Event Simulator. *Proceedings of the Winter Simulation Conference*, pages 776–782, 2002.

[5] J. Cowie. Towards realistic million-node internet simulations. *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[6] O. Dalle. OSA: an Open Component-based Architecture for Discrete-event Simulation. *Proceedings of the* 20^{th} *European Conference on Modeling and Simulation*, 2006.

[7] B. Delinchant, F. Wurtz, D. Magot, and L. Gerbaud. A component-based framework for the composition of simulation software modeling electrical systems. *Simulation*, 80:347–356, 2004.

[8] J. Earley. An efficient context-free parsing algorithm. *Communications of the Association of Computing Machinery*, 13:94–102, 1970.

[9] I. S. for Extended BNF. http://www.iso.org/iso/en/ cataloguedetailpage.cataloguedetail? csnumber=26153, 1996.

[10] I. for Software Research Irvine CA. Architecture description languages. *http://www.isr.uci.edu/architecture/adls.html*.

[11] I. Foster, C. Kesselman, and S. Tuecke. *Grid Computing, Making the Global Infrastructure a Reality*. John Wiley and Sons, 2003.

[12] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 2001.

[13] Y. Hu, G. Tan, and F. Moradi. Automatic SOM Compatibility Check and FOM Development. *Proceedings of 7th IEEE Distributed Simulation and Real-time Applications*, 2003.

[14] S. Kasputis and H. Ng. Composable simulations. *Proceedings of the Winter Simulation Conference*, 2000.

[15] J. Miller and P. Fishwick. Investigating ontologies for simulation modeling. *Proceedings of the 37th Annual Simulation Symposium*, pages 55–63, 2004.

[16] F. Moradi, P. Nordvaller, and R. Ayani. Simulation Model Composition using BOMs. *First Asia International Conference on Modelling and Simulation*, 2007.

[17] E. Page and J. Opper. Observations on the complexity of composable simulations. *Proceedings of Winter Simulation Conference*, 1:553–560, 1999.

[18] M. Petty and E. Weisel. A formal basis for a theory of semantic composability. *Proceedings of the Spring Simulation Interoperability Workshop*, 2003.

[19] M. Petty and E. W. Weisel. A composability lexicon. *Proceedings of the Spring Simulation Interoperability Workshop*, 2003.

[20] A. Samantaray and K. M. et al. Component-based modelling of thermofluid systems for sensor placement and fault detection. *Simulation*, 80:381–398, 2004.

[21] K. Shanmugan, W. LaRue, E. Komp, M. McKinley, G. Minden, and V. Frost. Block-oriented Network Simulator (BONeS). *Proceedings of IEEE Global Telecommunications Conference*, 3:1679–1684, 1998.

[22] D. Slama, K. Banke, and D. Krafziq. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, 2004.

[23] M. Spiegel, P. R. Jr., and D. Brogan. A case study of model context for simulation composability and reusability. *Proceedings of Winter Simulation Conference*, pages 437–444, 2005.

[24] C. Szabo and Y. M. Teo. An approach to syntactic composability and model reuse in composable simulation. Technical report, Department of Computer Science, National University of Singapore, 2006.

[25] A. Tolk. What comes after the Semantic Web - PADS Implications for the Dynamic Web. *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 55–62, 2006.

[26] S. Vadhiyar and J. Dongarra. A meta-scheduler for the grid. *Proceedings of 11th IEEE Symposium on High-Performance Distributed Computing*, pages 343–351, 2002.

[27] A. Verbraeck. Component-based distributed simulations. the way forward? *Proceedings of 18th Workshop on Parallel and Distributed Simulation*, pages 141–148, 2004.

[28] B. Ziegler, H. Prahofer, and T. Kim. *Theory of Modeling and Simulation*. Academic Press, 2000.