

CODES: An Integrated Approach to Composable Modeling and Simulation

Yong Meng TEO
Claudia SZABO

Asia Pacific Science and Technology Center
SUN Microsystems Inc.

November 2007
Singapore

CODES: An Integrated Approach to Composable Modeling and Simulation

Yong Meng TEO
Claudia SZABO

TECHNICAL REPORT
APSTC-TR-2007-04

Abstract

In component-based simulation, models developed in different locations and for specific purposes can be selected and assembled in various combinations to meet diverse user requirements. This paper proposes CODES (COMposable Discrete-Event scalable Simulation), an approach to component-based modeling and simulation that supports model reuse across multiple application domains. A simulation component is viewed by the modeller as a black box with an in- and/or out-channel. The attributes and behavior of the component abstracted as a meta-component are described using COML (COMponent Markup Language), a markup language we propose for representing simulation components. The integrated approach, supported by a proposed COSMO (COMponent-oriented Simulation and Modeling Ontology) ontology, consists of four main steps. Component discovery returns a set of syntactically valid model components. Syntactic composability is determined by our proposed EBNF syntactic composition rules. Validation of semantic composability is performed using our proposed data and behavior alignment algorithms. The semantically valid simulation component is subsequently stored in a model repository for reuse. As proof of concept, we discuss a prototype implementation of the CODES framework using queueing system as an application domain example.

Keywords: syntactic composability, semantic composability, model discovery, model reuse, simulation ontology

Email:
teoym@comp.nus.edu.sg



Asia Pacific Science and Technology Center
50 Nanyang Avenue, N3-01-C10
Singapore 639798

CODES: An Integrated Approach to Composable Modeling and Simulation *

Yong Meng Teo and Claudia Szabo

Department of Computing Science
National University of Singapore
Computing 1, Law Link
Singapore 117590

[teoym, claudias]@comp.nus.edu.sg

Abstract

In component-based simulation, models developed in different locations and for specific purposes can be selected and assembled in various combinations to meet diverse user requirements. This paper proposes CODES (COMposable Discrete-Event scalable Simulation), an approach to component-based modeling and simulation that supports model reuse across multiple application domains. A simulation component is viewed by the modeller as a black box with an in- and/or out-channel. The attributes and behavior of the component abstracted as a meta-component are described using COML (COMponent Markup Language), a markup language we propose for representing simulation components. The integrated approach, supported by a proposed COSMO (COMponent-oriented Simulation and Modeling Ontology) ontology, consists of four main steps. Component discovery returns a set of syntactically valid model components. Syntactic composability is determined by our proposed EBNF syntactic composition rules. Validation of semantic composability is performed using our proposed data and behavior alignment algorithms. The semantically valid simulation component is subsequently stored in a model repository for reuse. As proof of concept, we discuss a prototype implementation of the CODES framework using queueing system as an application domain example.

Keywords: syntactic composability, semantic composability, model discovery, model reuse, simulation ontology

1. Introduction

The modeling and simulation community has shown a growing interest towards building simulation models through model composition [11, 13, 28, 29]. Models developed using off-the-shelf components are appeal-

*A version of this technical report is published in the Proceedings of the 41st Annual Simulation Symposium, IEEE Computer Society Press, Ottawa, Canada, pages 103-110, 2008.

ing because of their shorter time to market and their flexibility in meeting diverse user needs [17]. Petty and Weisel [18] introduce two perspectives for studying composability, modeling and engineering. *Syntactic* (engineering) composability refers to the components' connections and communication. A composition that is syntactically valid guarantees correct and loose-coupled connections between components. Syntactic composability focuses on the implementation aspects of each simulation component. In contrast, *semantic composability* addresses whether the combined computation of the simulation model is semantically valid. Tolk [28] introduces additional composability levels, in particular pragmatic composability in which components are aware of the simulation context in which they are running. Syntactic composability can be viewed as an implementation oriented measure that reflects the components' interconnection and loose coupling. Chen and Szymanski [7] adopt a black box component view and use C++ templates to achieve syntactic composability. To demonstrate interoperability in supply chains simulations, Verbraeck [29] proposes a building block paradigm. A Fractal component model, proposed by Dalle [9], employs the notion of shared components to facilitate the use of the same common component across different components that encapsulate it. When we consider Petty and Weisel's definition of semantic composability with respect to data passing, we say that two components are semantically compatible if the data exchanged between them is correct and meaningful, and their behaviors are aligned, i.e., they receive inputs and produce outputs at correct moments in time. Two important aspects are worth highlighting. First, an algorithm that checks for semantic consistency need not use the component's implementation but a representation of it called *meta-component*, which encapsulates the component's behavior and attributes. An example of a meta-component oriented framework is BOM [11], which offers composability at the meta-component level but lacks semantic meaning and information. Secondly, the algorithm that verifies semantic composability prior to the simulation run can only give partial or incomplete answers since the notion of validity [2] assumes the presence of experts and existing simulation data or models for comparison [3].

Simulation composability and *model reuse* have proved more difficult to realize than initially envisaged [4]. The reuse of a simulation model [19] can be achieved by employing the model as it is, as a submodel within a larger application domain, or as a submodel across different application domains. Component-oriented frameworks exist for particular application domains [10, 21] but none offer cross-domain component integration or semantic composability. Approaches that generalize and share components across domains, both in-house and externally, increase the level of component reusability and facilitate the development of complex simulations at lower costs. Moreover, there is an increasing trend in using the Internet as an infrastructure for the discovery and (re)use of shared resources. By leveraging on Internet technologies such as peer-to-peer and web services, a web-based component-oriented simulation framework will

advance knowledge sharing to a wider simulation community. In this context, semantically valid simulation components and simulations have an increased market value and offer a quantum leap in the capability and credibility of component-based simulation models.

An important issue in addressing composability, in particular semantic composability, is expressing domain or component knowledge in an unambiguous, standardized format. An *ontology* is an organized knowledge representation to capture object information in a particular domain [26], in formats readable by humans and computers alike. Ontologies are conceptual models that capture and explain the vocabulary used in semantic applications guaranteeing communication free of ambiguities [6]. When applied to the modeling and simulation domain, ontologies facilitate model discovery and integration and the development of formal methods for simulation and modeling [14, 23]. Ontologies can be used to express syntax and semantics to facilitate communication and allow for automated semantic checking as well as to express the resource discovery request and determine whether the discovered model is reusable. Recent work in simulation ontologies includes DEMO and PiMODES. DEMO (Discrete Event Modeling Ontology) [14] focuses on exhaustively classifying simulation models with respect to their particular worldview. However, the DEMO ontology lacks detail when describing model attribute and behavior. A model is viewed as having ModelComponent(s) and functioning according to a state-based ModelMechanism, but little detail is given beyond that. The PiMODES ontology [23] focuses on models adhering to the process interaction worldview. It includes sets of classes for basic concepts, activities and control flow relationships of activities with the purpose of interchanging between proprietary models that adopt the process interaction worldview. However there does not exist an ontology to capture the component oriented aspect of the simulation domain. Ideally an ontology should focus on the description of simulation component to facilitate semantic validation of compositions, as well as to support component discovery and reuse.

The contributions of this paper follow.

1. We propose CODES, a systematic and complete approach to component-based simulation model development. We address the four main steps in component-based simulation development, namely, *model discovery*, *model reuse*, *syntactic* and *semantic composition*. We propose paradigms and hierarchies to facilitate simulation composition from CODES components, and, in contrast to current approaches, advance component reuse across simulation application domains.
2. We propose COSMO, a component-oriented ontology that describes components and compositions. The COSMO ontology facilitates checks for semantic correctness, component discovery and reuse.
3. Based on the COSMO ontology and the CODES paradigms, we propose an algorithm for semantic

checking of compositions that consists of two stages: *data alignment* and *behavior alignment*. We implement a prototype of CODES and present an example of simulation model development to validate and demonstrate the scalability of our framework.

This paper is organized as follows. Section 2 presents the CODES framework design, highlighting our approach to syntactic and semantic composability, including the COSMO ontology. In Section 3 we present the CODES prototype and demonstrate its scalability by presenting the process of adding a new application domain. We compare the CODES framework with other proposed frameworks in Section 4. Finally, Section 5 concludes this paper.

2. CODES Component Framework

This section presents an overview of our CODES framework and discusses in greater details our proposed component-oriented COSMO ontology, components and their higher-level representation as meta-components, and our approach to semantic composability. Our proposed approach to syntactic composability using EBNF can be found in [27] and will not be elaborated.

2.1. Overview

CODES (Composable Discrete-Event scalable Simulation) is a hierarchical component framework to support component-based modeling and simulation. It addresses four key requirements in component-based model development and simulation, namely, *component discovery*, *model reuse*, *syntactic* and *semantic* composability. CODES is based on our proposed *component-connector paradigm* to which all CODES application specific base components and model components must comply. In the component-connector paradigm, a *component* is viewed as a black box with an in- and/or an out-channel. Components are interconnected by *connectors* which perform message and data passing among them. Messages leaving components are timestamped and the connector guarantees FIFO delivery of messages to the destination components. Connectors include *one-to-one* for connecting two components, *many-to-one* for multiplexing 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 [25] of components and together with our proposed composition rules specified in EBNF, provides for syntactic composability [27]. Syntactic consistency check is applied first to reduce the set of potential models before the checking of semantic compatibility.

A CODES component is modelled as a meta-component on top of the actual component implementation.

The meta-component describes the *attributes* and *behavior* of a component and is used in the framework for model reasoning and discovery. The component behavior describes the data that it receives and outputs as a set of states. The transitions between states can be defined as a set of triggers expressed in terms of input, time and conditions. The data interchanged between components is attached with a set of semantically enriched constraints to validate the communication. Thus, we say that two components are *semantically compatible* iff they are aligned with respect to *data exchange*, (i) data passed between the components is correct and meaningful to both parties, and *behavior*, (ii) data exchange does not hinder the advancement of the components towards a final state or a state that produces *valid output*. Based on the meta-component representation and COSMO, an ontology we proposed for component-oriented simulation, we define semantic consistency and design semantic checking algorithms in the CODES framework accordingly. The COSMO ontology facilitates more precise description of the component discovery request, as well as provides a measure for the reuse potential of components in a given simulation context.

Besides achieving syntactic and semantic composability, a component-based simulation framework must also support component discovery, reuse of developed simulation models, as well as integration with off-the-shelf simulation components. To be scalable, component discovery must be achieved in a distributed context, where component providers and consumers reside in different administrative domains. A component discovery service must handle complex component queries in a time constrained environment, while guaranteeing reliability and scalability. Our solution to model reuse allows for reuse of a simulation model both “as-is” (as a simulator), and as a model component in a larger simulation model. The reuse procedure is closely tied with the discovery service which in turn relies on how model components are saved and represented in the model repository. The subject of search queries can be both for standalone components and for model components or simulators. In the case of standalone components only semantic information regarding the component’s attributes and behavior is provided by the user, and the discovery service employs the COSMO ontology to find relevant matches. When a search query arrives for a model component the discovery service searches initially for syntactically compatible models, i.e. models that contain a similar sequence of components to that presented in the query. Next, a semantic search using the COSMO ontology and the behavior and attributes specified by the user is performed on the syntactic compatible models. Ideally the discovery service should provide a measure of the relevance of the discovered model components with respect to the query and the larger composition into which they will be integrated. As with all semantic discovery services, the user can be involved in providing feedback about the suitability of the results, and also to refine the search query when suitable results are not found.

Figure 1 presents the CODES simulation model development process. A CODES user creates a concep-

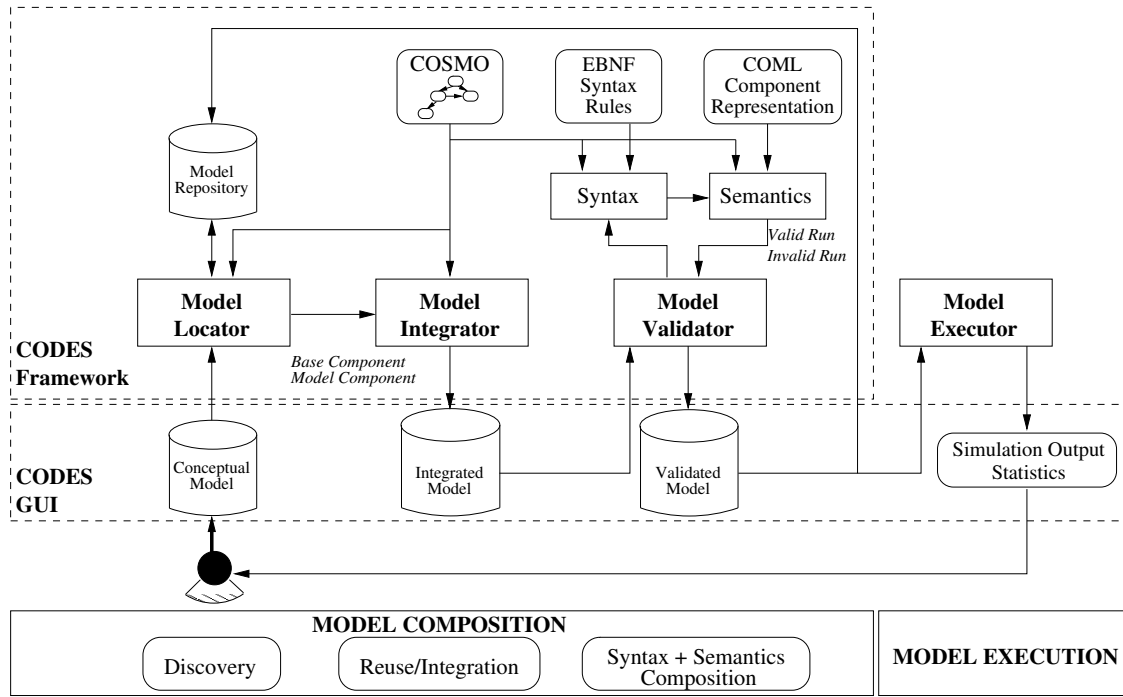


Figure 1. Component-based Simulation Model Development

tual model through the use of the CODES GUI. Based on the conceptual model and the COSMO ontology, the *Model Locator* formulates a set of search criteria and sends a resource discovery request to the *model repository*. A model repository is organized as a hierarchical database of annotated (or indexed) models, model components and application specific base components. The search results are evaluated by the *Model Integrator* which performs relevant assessment of the models or model components using the ontology, and if a perfect match is found, composes them into the integrated or discovered model. The discovered model undergoes syntactic and semantic checks by the *Model Validator*. The Model Validator employs syntax rules expressed in EBNF for syntactic checking [27]. The COSMO ontology and companion component representation are used in semantic checking. If the Model Validator returns a positive answer, the validated model is forwarded to the *Model Executor* which provides simulation output and statistics. Furthermore, the validated model is placed into the model repository for reuse. Besides specifying the conceptual model, user input is necessary in at most two points in the simulation flow. First, if a perfect match for the conceptual model is found, the user must specify the attribute values for all components in the discovered model. Secondly, if a match is not found, the user can select a component or model components from the candidates provided (if any) or may choose to implement new component(s).

2.2. COSMO Ontology

COSMO (COmponent Simulation and Modeling Ontology) is a proposed ontology that focuses on describing component-oriented simulation within and across application domains. COSMO semantically enriches the description of model components to support model discovery, model reuse and the semantic composability check of the discovered models. The ontology consists of sets of classes to describe simulation components and the compositions of simulation components.

The hierarchies in the COSMO ontology span two main directions, as it can be seen in Figure 2. To achieve generality across application domains and at the same time support specific application domain requirements, we include first an application domain oriented component hierarchy. The component repository consists of shared components common to all application domains, and shared components specific to each application domain. An application domain defines its own specific pool of *BaseComponents*. Composed models are placed in the component repository as *ModelComponents*. The other set of classes describes

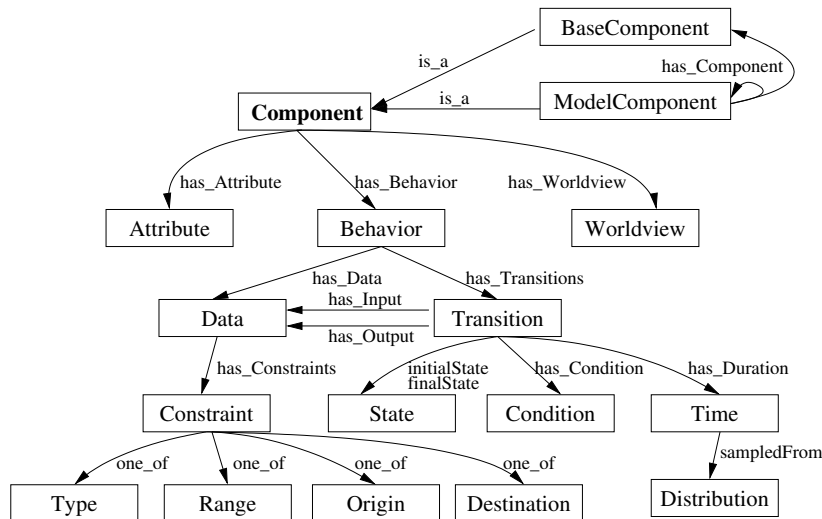


Figure 2. COSMO Ontology Structure

components with respect to their *attributes* and *behavior*. We assume that irrespective of the simulation component's implementation and worldview, its behavior can be represented as a finite state machine initially provided by the component creator. Transitions in the state machine from an initial to a final state are triggered by an arrival event or an elapse in a time interval. The final state can be determined by some conditions on the component's attributes and the transition may produce output. The classes for attribute, behavior, worldview, transition, state, data, condition as well as simulation concepts such as time, distributions, etc. are defined in the ontology.

Research in Semantic Web [5] has spearheaded the development of knowledge representation languages

such as XML and RDF to meaningfully represent web resources in ontologies. The Web Ontology Language (OWL) [15] has evolved as the standard language to represent ontologies, with three flavors: OWL Lite offers hierarchies and basic constraints; OWL DL, based on Description Logics [1], is computationally complete and decidable; and OWL Full is the most expressive of the three but with no guarantees on completeness. The COSMO Ontology is written in OWL DL and has been developed using Protégé [20], a widely used ontology builder. Figure 3 presents a snapshot of the asserted COSMO class structure in Protégé.

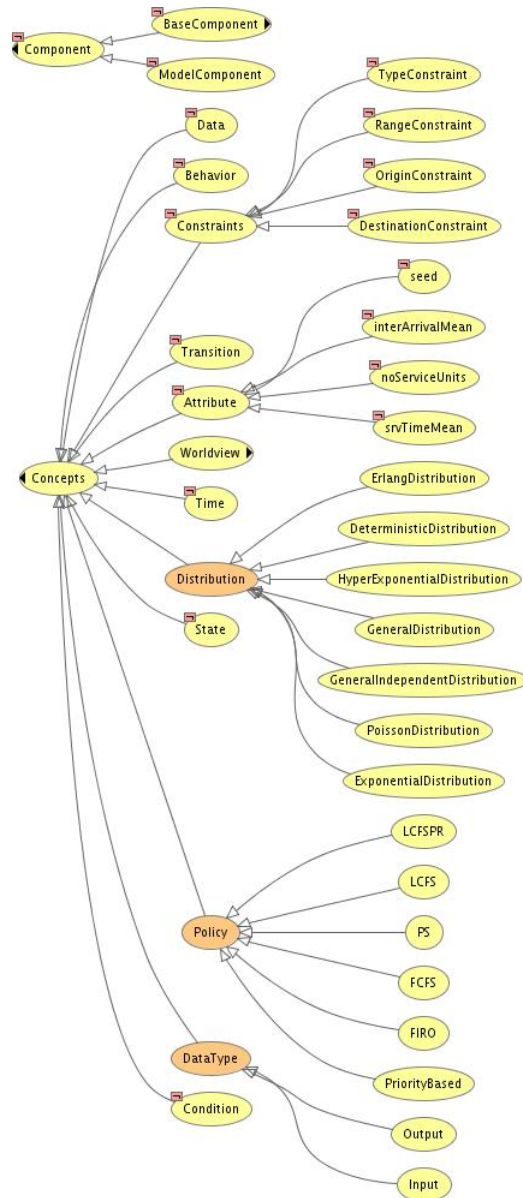


Figure 3. COSMO Asserted Ontology Class Structure

2.3. Component Representation

A CODES component, viewed by the modeller as a black box, is described internally as a meta-component to support semantic checking and discovery. A meta-component describes the attributes and behavior of a simulation component. Mandatory attributes are common attributes specified by the framework for all components, and specific attributes are particular to each component. The behavior of a component is modelled as a finite state machine starting from an initial state and proceeding to a final state. Transitions between states occur upon the component's receiving input, may take a certain time and can produce output. A transition may change the values of some specific component attributes. The final state can be determined by evaluating conditions expressed in terms of component specific attributes. It is important to highlight here that the state machine and the changes on the attributes may not occur per se in the component's implementation. The meta-component captures information provided by the component creator and although it is assumed correct, there is no formal guarantee to that.

To ensure that component communication is meaningful, components have constraints attached to the data they can receive or send. The constraints include the *type* of data, *range* of its values, *origin*, *destination* or a specific *time* interval. For example, a component can only receive/send data of a given type and in a given interval, arriving from a specific semantically enriched origin, departing to a semantically enriched destination or arriving at a specific local time. By semantically enriched constraints we mean that the value of the constraint is a *class* in the COSMO ontology and the component to which the data goes or comes from is a type of that class or of one of its subclasses. Data alignment is discussed in Section 2.4.

More formally, a *base component* C_i is represented by the tuple:

$$C_i = \langle R, A_i, B_i \rangle$$

where R represents the required (mandatory) attributes that are common to all components, A_i denotes component specific attributes, and B_i denotes component behavior. A component behavior is represented as follows:

$$[I_l]S_p[\Delta t] \xrightarrow{Cond_n} S_t[O_l][A_m]$$

where I_l is the set of input data; S_p is the current state; Δt is the translation duration; $Cond_m$ defines the condition(s) for the state transition; S_t is the next state; O_l is the set of outputs after the state change; A_m is the set of attributes and their values that are involved in the state changes.

To represent simulation models in a machine readable form we propose COML, a markup language for

representing base and model components. COML is described in XML Schema using entities in the COSMO ontology as a starting point. CODES components are described using XML, which is checked against the COML schema. Figure 4 presents excerpts from the COML Schema for base components. COML also provides a schema for representing the data that flows between components. When a component sends output data, it is wrapped by the component using the meta-data format provided by COML. Conversely, the data needs to be unwrapped first upon receiving it. This guarantees that communication between components is performed in a correct and standardized format.

2.4. Syntactic and Semantic Composability

Syntactic composability is defined as the actual implementation of composability, where loose coupled components interoperate [3]. In the CODES framework, this is done through the implementation of the black box and component-connector paradigm. Syntactic composability is taken one step further by defining a two level hierarchical composition grammar expressed in EBNF [27]. The first level of the composition grammar describes composition rules for all components in the CODES framework and is a formalization of the component-connector paradigm. The rules from the first level are specialized to contain specific application domain composition rules (see Figure 7(b)) and form the second level. The composition grammar is built by domain experts and is used to verify that the composition is syntactically valid.

Semantic composability refers to simulation components forming a meaningful and computationally valid composition. A CODES simulation component is described as a black box implementation, with a meta-component containing a description of attributes and behavior attached to it. The component's behavior is expressed as a function of the input that it receives and the output that it produces. Thus, we relate semantic composability to this data exchange. Namely, the components should be able to communicate correctly, understand the simulation context and produce a valid output as a whole. We say that simulation components are *semantically compatible* and form a *semantically viable composition* iff the data exchange among neighboring components and the components' behaviors are aligned, and the composition produces valid outputs. The data that a component sends and receives is attached with a set of constraints. The types of constraint include origin, destination, time, type and range, and the list can be extended. In order for the components to be aligned with respect to data exchange, all data constraints between neighboring components must be met. Components' behavior is aligned if the data exchange does not hinder the components' transitions to a final state or a state that produces output. Thus the data exchange between components is such that no component hangs during the simulator execution. In order to declare the composition valid, the simulation must be run and its results analyzed by experts or compared to other correct results. However,

```

<complexType>
  <sequence>
    <element name="mandatoryAttributes" type="spec:mandAtt"/>
    <element name="specificAttributes" type="spec:specAtt"/>
    <element name="data" type="spec:dataType" maxOccurs="unbounded"/>
    <element name="behavior" type="spec:behaviorType"/>
  </sequence>
</complexType>

<!-- Restriction- initial and final state name for each transition should be from the set of states -->
<key name="stateName">
  <selector xpath="behavior/states/state"/>
  <field xpath="@name"/>
</key>
<keyref name="ref1" refer="spec:stateName">
  <selector xpath="behavior/transitions/transition/initial"/>
  <field xpath="@name"/>
</keyref>
<keyref name="ref2" refer="spec:stateName">
  <selector xpath="behavior/transitions/transition/final"/>
  <field xpath="@name"/>
</keyref>
...
</element>

<complexType name="mandAtt">
<sequence>
  <element name="name" type="string"/>
  <element name="type" type="string"/>
  <element name="author" type="string"/>
  ...
</sequence>
</complexType>

<complexType name="specAtt">
<sequence>
  <element name="attribute" maxOccurs="unbounded">
    <complexType>
      <sequence>
        <element name="value" type="string"/>
        <element name="description" type="string" minOccurs="0"/>
      </sequence>
      <attribute name="name" type="string"/>
    </complexType>
  </element>
</sequence>
</complexType>

<complexType name="behaviorType">
<sequence>
  <element name="inputs" type="spec:datas" maxOccurs="unbounded"/>
  <element name="outputs" type="spec:datas" maxOccurs="unbounded"/>
  <element name="states" type="spec:stateType" maxOccurs="unbounded"/>
  <element name="durations" type="spec:timeIntType"/>
  <element name="attributes" type="spec:modifAtt" maxOccurs="unbounded"/>
  <element name="conditions" type="spec:conditionsType" maxOccurs="unbounded"/>
  <element name="transitions" type="spec:transitionsType"/>
</sequence>
</complexType>

```

Figure 4. Component Representation in COML

this is not possible when doing pre-run semantic composability validation on meta-components. Nevertheless, the composition builder can be involved in the process of semantic checking by providing information about valid data at user-specified *validity points*. This valid data is expressed in terms of data constraints and serves as starting points in the checking algorithm. For each validity point, the semantic checking algorithm uses the meta-components' behavior representation to determine if there exists a simulation run capable of producing the output specified in the validity point. However, if there exists a run to produce the desired valid data, the algorithm cannot guarantee that the simulation run actually takes place. This guarantee is provided by the second part of the behavior alignment checking algorithm, which verifies that the components' individual run does not hang throughout the simulation run. If there is no simulation run to produce the valid data, the user can modify the validity point and components' attributes, as well as to replace components with equivalent or poorer matches.

Our algorithm for semantic checking is divided in two stages: check the *data alignment* of components and verify *behavior alignment*. Figure 5 shows our semantic checking process. The first stage checks for

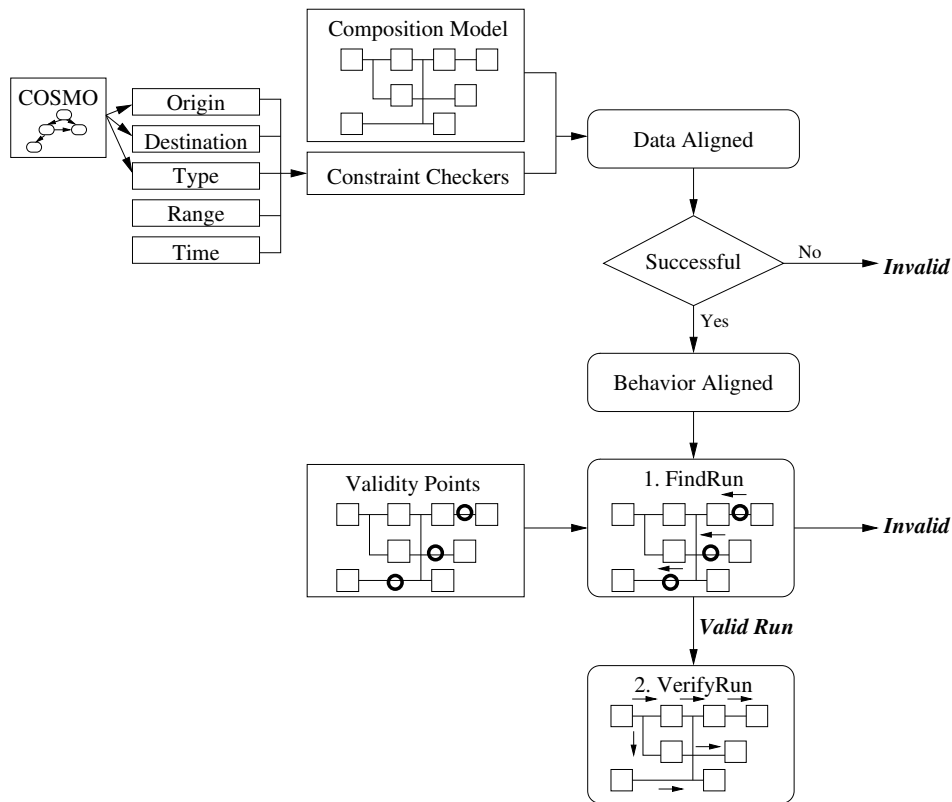


Figure 5. CODES Semantic Checking

data alignment in all components, by calling $DataAligned(left, right)$ for every pair of connected *left* and *right* components. For every $(left, right)$ pair, and for every type of constraints, the modularized constraint checker calls the respective constraint modules to verify that the constraints on *left*'s output subsume the

constraints on *right*'s input. For example, for a constraint of type "range", verified by the *RangeConstraint* module, the range specified for the *left* component must be included in or equal to the *right* component's range. Similarly for constraints of type "origin" and "type", where the *left* component must be of the same class or a subclass of the class mentioned in the *right* component's data constraint, and the type of the *left* output data should be a subtype of the *right*'s data's specified type respectively. The *OriginConstraint* module checks to see if the origin specified by the input data of the *right* component is consistent with the *left* component. For this, the representation of the *left* component is translated into OWL and the constraint reasoner checks to see if it is of the type of the class specified by the input data of the *right* component. If it is not, the algorithm concludes that the components are not aligned and the semantic checking algorithm returns a negative answer. The semantic checking algorithm pseudo-code is shown in Figure 6.

Figure 6. CODES Semantic Checking Algorithm

<pre> boolean SemanticChecking (CompList l) if (CheckData(l)) return BehaviorAligned(pointList); boolean BehaviorAligned(ValidPoint [] pointList) sum = 0; for ValidPoint p : pointList Component[] comps = getNeighbors(p); if (comps == null) return false ; aux = FindRun(comps,p); if (aux == 0) return false ; sum += aux; if (sum != pointList .length) return false ; return true ; </pre>	<pre> boolean DataAligned(Component left ,Component right) for ConstrChecker Constraint : constraintList if (! Constraint (left , right)) return false ; return true ; boolean FindRun(Component[] comps, Point p) if (existsCompOnlyOutChannel(comps)) return true ; sum = 0; for Component c : comps if (visited (c)) {sum++; continue;} InputData input = getInput (c,p); if (input==null) {sum++; continue;} Point p1 = getPoint (input ,c); Component[] cps = getNeighbors(p1); if (cps==null) {sum++; continue;} if (sum==comps.length) return false ; return FindRun(cps,p1); </pre>
--	---

If the data alignment stage is successful, the behavior alignment stage can proceed. The behavior alignment algorithm consists of two stages: check for the existence of valid output, and that no component's behavior hinders others. For the first stage, the *BehaviorAligned* function is called with the user specified validity points. Using the GUI, the user specifies validity points on the connectors that link components. Each validity point comprises of a set of constraints on the data that flows through the connector to which it is attached, and thus from the component(s) on the connector's left side to the component on the right side. For all validity points, we first determine the neighbouring component(s) that send output through these points. Next, the output data of the neighboring components is verified against the valid data to check if the constraints are met, by the same procedure as in the data alignment phase. If the constraints are not met, the algorithm returns an *Invalid* message, concluding that the composition is not semantically viable. The

FindRun method is called for the set of outputs that matches the constraints on the valid data and respective components to determine if there exists a run in the composition that produces the outputs. For every component, the method tries to determine what are the inputs that lead to the production of the said outputs, obtaining pairs of $(input, output)$. If such $(input, output)$ pair does not exist, the algorithm returns an *Invalid* message. The components connected to the current one are next evaluated to determine if they produce the input needed by the component. Note that there is no need for constraints matching at this point, since it has already been done by the data alignment phase. The method continues recursively for the components connected to the current one, until either a matching component with only an *out* channel has been found, or all components have been visited. If at least one component with only out channel that produces the desired output is found, the algorithm returns a *Valid* message, concluding that there is a simulation run to produce the valid data, and outputs the $(input, output)$ pairs.

If a valid run is found, the behavior alignment algorithm's second stage (*VerifyRun*) checks that no component run inhibits the run of other components. This stage is difficult to complete because of the time constraints that may be put on the transitions. Output from a simulation component is produced after that component changes state. The state change happens as a result of an input from another component, correlated with the passing of time and some conditions on the specific attributes of the component. This means that for a component to produce output or reach a final state (or both) it is important not only for the inputs to be of a given type and meet data alignment constraints, but also that they come in a specific order and at certain time intervals, usually sampled from some distributions. Thus, even though two components are aligned with respect to data exchange, one component may need output from the other at a specific stage of its run, and may wait for it indefinitely if it does not arrive or arrives too late or too early. In order to complete the second stage, a mock-up run of the composition involving the meta-components (and thus a meta-simulator) should be executed. However, this approach is computationally expensive and thus undesired, since the semantic checking process is inherently time constrained in a user-oriented composition process. We limit for now the implementation of the behavior alignment check to the first stage.

3. CODES Prototype

The CODES framework and GUI are implemented in Java. The CODES framework prototype implementation is modularized, following the structure presented in Figure 1. We use the Jena [12] API, a semantic web framework for Java, and the Pellet [24] reasoner for constraint checking using the COSMO ontology. The CODES framework is built on top of the Scalable Simulation Framework (SSF) [8] which provides a

Java API for building discrete-event simulations. 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 a CODES component.

In web-based simulation, model repositories are generally distributed and components may be available as web services. In this setting, the CODES framework provides for scalability with respect to the number of application domains and the number of components, since until the simulation execution phase all reasoning is done on COML representations and thus components need not be available until runtime. The current version of the CODES framework must have the components available locally for the simulation run. The next version of the CODES framework aims at fully providing for web-based distributed repositories and components.

This section discusses the steps to develop a simulation model using our CODES GUI as a front-end and our framework as a back-end. We describe the process of adding a new application domain to the framework and how a simulation model is developed. Next, we present an example of how our semantic checking algorithm determines the semantic consistency of a composition.

3.1. Adding a New Application Domain

Assuming we start with a new version of the framework, it is the job of trusted users or administrators to add new application domains to CODES. Adding a new application domain involves extending the CODES ontology by providing descriptions of the domain’s base components and extending the CODES composition grammar by adding composition rules (if any) specific to that application domain. After a new application domain has been added, models can be built using its base components. For the current CODES prototype it is only possible to build models for specific application domains using their base and model components. The next version will support cross-domain integration of components.

For illustration, we use the Queueing Networks application domain, with three base components, namely, source, server, and sink. *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*.

The addition of the queueing networks base components to the CODES repository is reflected in the COSMO Ontology as shown in Figure 7(a). *QNBaseComponent* is added as a new subclass of the BaseCom-

ponent class, with *Source*, *Server*, and *Sink* as its subclasses. Besides having attributes and behavior like their superclass, the Source component must have an *interArrivalTimeMean* attribute and a arrival time *Distribution*, and the Server component a *serviceTimeMean* attribute, a service time *Distribution*, a *noServiceUnits* attribute, and a service *Policy*. *has_IATime*, *has_SrvTime*, and *has_ArrivalTimeDistr*, *has_ServiceTimeDistr* are subproperties of *has_Attribute* and *has_Distribution* respectively. The addition of a new application domain is reflected by extending the CODES composition grammar to include queueing networks composition rules, as shown in Figure 7(b).

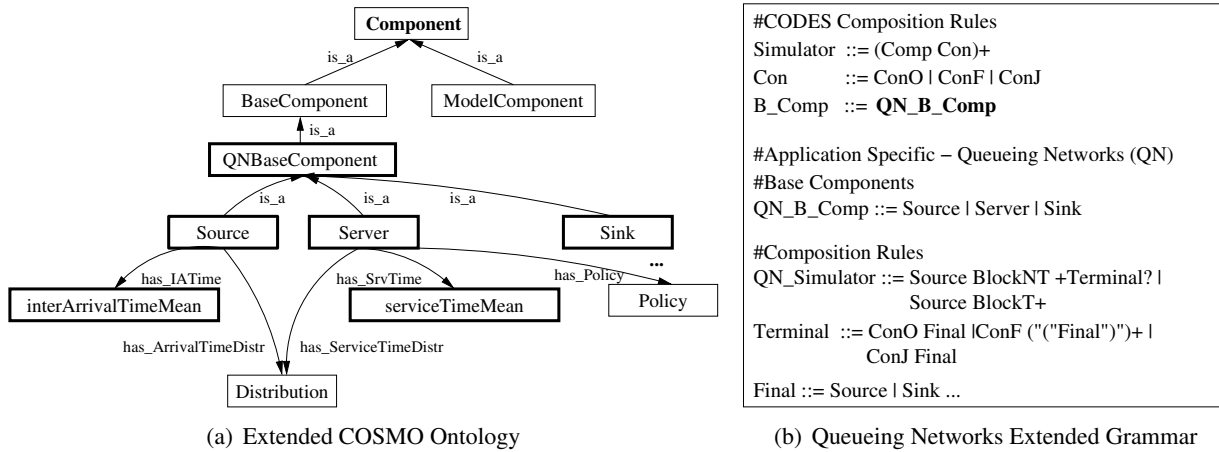


Figure 7. CODES Queueing Networks Application Domain

We assume that the model repository contains one implementation for the source, server and sink base components called *source1*, *server1*, and *sink1*, respectively. Each component repository must have a meta-component attached to it, with attributes and behavior represented in COML. The component representation is verified against the COML schema when the component is added to the model repository. Figure 8 shows part of the attributes and the state machine for the *server1* component. The *server1* component requires input data of type *double* using the wrapper class *Job*, with values in the range of $[10,35]$ and with *origin* a component of type *Source* or *Server*. Following state transitions through five states, the component produces output in the range $[14,18]$ with *destination* a component of type *Sink* or *Server*. The constraints on the input and output data for component *server1* can be expressed respectively as $I_{server1} = \{type = double, range = 10:35, origin = Source|Server\}$, and $O_{server1} = \{type = int, range = 14:18, destination = Server|Sink\}$.

3.2. Building a Simulation Model

Suppose a user builds the conceptual model for a single server queueing system using the CODES GUI. This is done by simple drag'n drop operations on the base components' icons presented in Figure 9 to a drawing panel in the GUI and subsequent connections (symbolizing connectors) of these icons. At this

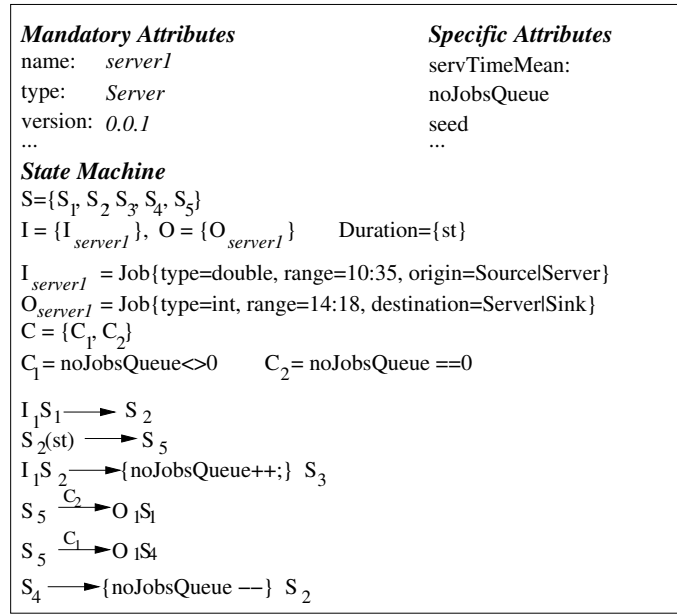


Figure 8. Meta-Component for Component *server1*

point the icons represent only conceptual notions of the base components, without any implementation or representation of attributes and behavior behind them. It is only after the discovery process by the Model Locator that these materialize.

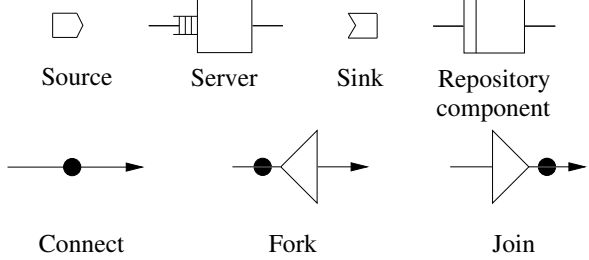


Figure 9. CODES Component and Connector Icons

In the CODES framework, the Model Locator module performs semantic queries on the repository to locate existing implementations for each of the base components specified in the conceptual model, using user specified attributes. We assume that perfect matches are found for each base components, *source1*, *server1*, and *sink1* for the source, server and sink respectively. The user then sets the attribute values for the discovered components as well as the constraints for one validity point as shown in Figure 10 and upon the completion of this stage the Model Validator module is called in the framework back-end.

First, the syntax checker module verifies that all components are connected correctly, for example, there is no component with an out channel left unconnected. Then the model's production string is checked against the CODES composition grammar [27]. The model's production string is a linear arrangement of the components' types according to their position on the graphical screen. For the single server model the production string is

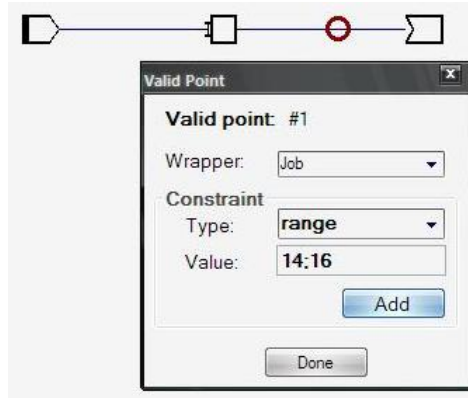


Figure 10. Adding Valid Point Constraints Using the CODES GUI

$MM1_QN = Source\ ConO\ Server\ ConO\ Sink$

and returns a positive answer from the grammar checker. Thus the checking can proceed to the semantic checking phase.

The data alignment checking phase verifies that the data constraints for all components are met. The constraints for the repository components are $O_{source1} = \{type = int, range = 10:20, destination = Server\}$ for component's source1 output,

$I_{server1} = \{type = double, range = 10:35, origin = Source|Server\}$ for component's server1 input, and

$O_{server1} = \{type = int, range = 14:18, destination = Server|Sink\}$ output, and

$I_{sink1} = \{origin = Server\}$ for component's sink1 input.

The constraint checker takes every pair of connected components and verifies that the constraints on input and output data match. In the single server model, there are two pairs of connected components, (*source1*, *server1*) and (*server1*, *sink1*), with six constraints between them. For the constraints of type *origin* and *destination*, e.g. for $O_{source1}$ with *destination* = *Server*, the constraint checker must verify that the connected component, i.e. *server1*, is a type of the class specified by the constraint, i.e. *Server*. The COML file for *server1* is translated into a RDF file and querying for the *RDFS.type* property is done on the COSMO ontology using the Jena libraries. The constraints of type *range* are checked as interval inclusions, e.g. $[10, 20] \subseteq [10, 35]$ between *source1* and *server1*. The *type* constraints are checked based on a subsumption type table, e.g. $int \subseteq int$, and $int \subseteq double$, etc.

Upon the completion of the data alignment check, behavior alignment checking can proceed. For this model and the validity point

$$VP_1 = \{origin = Server, range = 14:16, type = double\},$$

there exists a valid run, with the *(input, output)* pairs as

$$(O_{server1}, VP_1), (I_{server1}, O_{server1}), (O_{source1}, I_{server1}), \text{ and } (-, O_{source1}).$$

The validated model can now be ran using for each component the attributes previously set by the user and the user receives simulation output from the GUI.

4. Related Work

To demonstrate the feasibility of interchanging between simulation packages, Lacy et al. develop the PIMODES [23] ontology which describes models that adhere to a process interaction worldview. Simulation models can be created as PIMODES models and then translated into executable models for commercial simulation software, and the reverse is also possible, but there is no provision for component-based simulation. FAMOS [16] is a framework for process driven applications, including process interaction simulation, developed to facilitate sharing between process oriented applications using ontologies. FAMOS employs the Neutral Process Representation Languages to overcome the setting when the semantic context and intentions of the system and application designers is not available, by providing effective translations for applications to share information.

In [22] Silver et al. discuss the issues in building ontology driven simulations, in which the creation of simulation models is driven by domain specific ontologies, and present the Ontology Driven Simulation (ODS) design tool. Concepts in the domain ontology are mapped to simulation concepts in the DEMO [14] ontology. Reusability across different domains and ontologies is achieved by representing the models in the DEMO ontology and then translating to and from different simulation packages. The DEMO ontology is also used in determining class inconsistencies in the model, for example when similar concepts in the domain ontology are mapped to disjoint concepts in the DEMO Ontology. Similar to ODS, we use the COSMO ontology to verify semantic consistency. However, instead of focusing on mapping concepts from different domain ontologies to a simulation ontology, we achieve this directly in the component-oriented COSMO ontology together with our proposed composition grammar. COSMO is both component and application domain oriented and is naturally scalable. Thus, the process of determining the suitability of a component in a composition is simplified to a more straightforward query operation by a logic reasoner working with component representations in COML. Furthermore, we provide an integrated approach to component-based simulation development in which the consistency of the composition and the reliability of the discovery

service are the main issues.

5. Conclusion

We present CODES, a scalable approach to component-based simulation to facilitate web-based simulation and integration of components across multiple application domains. The integrated approach consists of four main steps, namely *component discovery and reuse*, *syntactic composability checks*, and *semantic composability validation*. To facilitate component discovery and reuse, we propose the COSMO ontology, a component-oriented ontology in which simulation components with *attributes* and *behavior* are classified as *base components*, specific to an application domain, and *model components*, span multiple application domains. CODES components are viewed as black boxes interconnected by connectors that pass data in the form of messages between components. The CODES component-connector and black box paradigms, as well as a hierarchically scalable composition grammar with composition rules specific for each application domain, determine the syntactic composability of a component with its neighboring components. To achieve semantic composability validation, a CODES black box component is specified as a meta-component in COML, a markup language we propose to express the attributes and behavior of a component. The behavior of a component is expressed as a state machine with inputs and outputs. Components are semantically compatible if the data exchanged between them is correct and meaningful and their behaviors are aligned, i.e., they receive inputs and produce outputs at correct moments in time. We present a semantic checking algorithm that employs the COSMO ontology to check for data alignment, and together with user specified validity points, checks for behavior alignment. We demonstrate the feasibility of our approach to component-based simulation by an example of the simulation model development using the prototype implementation of the CODES framework and GUI. Future work includes refinements of the behavior alignment algorithm. We are also exploring the reuse of partially valid semantic components. To assist users in selecting a partially compatible model component, this will require a quantitative measure of the degree of semantic compatibility of components.

References

- [1] F. Baader and W. Nutt. *The Description Logics Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, UK, 2003.
- [2] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice Hall, USA, 2005.
- [3] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. In Search of the Philosopher’s Stone: Simulation Composability Versus Component-Based Software Design. In *Proceedings of the Fall Simulation Interoperability Workshop*, Orlando, USA, 2004.
- [4] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. The Computational Complexity of Component Selection in Simulation Reuse. In *Proceedings of Winter Simulation Conference*, pages 2472–2481, Orlando, USA, 2005.

- [5] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 28–37, 2001.
- [6] K. Breitman, M. Casanova, and W. Truszkowski. *Semantic Web: Concepts, Technologies and Applications*. Springer Verlag, London, UK, 2007.
- [7] G. Chen and B. Szymanski. COST: A Component Oriented Discrete Event Simulator. In *Proceedings of the Winter Simulation Conference*, pages 776–782, San Diego, USA, 2002.
- [8] J. Cowie. Towards Realistic Million-Node Internet Simulations. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2129–2135, Las Vegas, USA, 1999.
- [9] O. Dalle. OSA: An Open Component-based Architecture for Discrete-event Simulation. In *Proceedings of the 20th European Conference on Modeling and Simulation*, Prague, Czech Republic, 2006.
- [10] 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.
- [11] P. Gustavson and L. Root. Object Model Use Cases: A Mechanism for Capturing Requirements and Supporting BOM Reuse. In *Spring Simulation Interoperability Workshop*, Orlando, USA, 1999.
- [12] Jena - A Semantic Web Framework for Java. <http://jena.sourceforge.net>, 2002.
- [13] S. Kasputis and H. Ng. Composable Simulations. In *Proceedings of the Winter Simulation Conference*, pages 1577–1584, Orlando, USA, 2000.
- [14] J. Miller and P. Fishwick. Ontologies for Modelling and Simulation: Issues and Approaches. In *Proceedings of the Winter Simulation Conference*, pages 259–264, Orlando, USA, 2004.
- [15] OWL Web Ontology Language Overview. <http://www.w3.org/tr/owl-features/>, 2004.
- [16] B. Perakath, K. Akella, K. Malek, and R. Fernandes. An Ontology-Driven Framework for Process-Oriented Applications. In *Proceedings of the Winter Simulation Conference*, pages 2355–2363, Orlando, USA, 2005.
- [17] M. Petty and E. Weisel. Basis for a Theory of Semantic Composability. In *Proceedings of the Spring Simulation Interoperability Workshop*, Orlando, USA, 2003.
- [18] M. Petty and E. W. Weisel. A Composability Lexicon. In *Proceedings of the Spring Simulation Interoperability Workshop*, pages 181–187, Orlando, USA, 2003.
- [19] M. Pidd. Simulation Software and Model Reuse: A Polemic. In *Proceedings of the Winter Simulation Conference*, pages 772–775, Washington, USA, 2004.
- [20] Protege Ontology Editor. <http://protege.stanford.edu>, 2004.
- [21] K. Shanmugan, W. LaRue, E. Komp, M. McKinley, G. Minden, and V. Frost. Block-oriented Network Simulator (BONeS). In *Proceedings of IEEE Global Telecommunications Conference*, pages 1679–1684, Sydney, Australia, 1998.
- [22] G. Silver, O. A.-H. Hassan, and J. Miller. From Domain Ontologies to Modeling Ontologies to Executable Simulation Models. In *Proceedings of the Winter Simulation Conference*, page (to appear), Washington, USA, 2007.
- [23] G. A. Silver, L. Lacy, and J. A. Miller. Ontology Based Representations of Simulation Models Following the Process Interaction World View. In *Proceedings of the Winter Simulation Conference*, pages 1168 – 1176, Monterey, USA, 2006.
- [24] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5:51–53, 2007.
- [25] D. Slama, K. Banke, and D. Krafziq. *Enterprise SOA: Service-Oriented Architecture Best Practices*. Prentice Hall, USA, 2004.
- [26] J. Sowa. *Knowledge Representation: Logical, Philosophical and Computational Foundations*. Brooks/Cole, Pacific Grove, USA, 1999.
- [27] C. Szabo and Y. Teo. On Syntactic Composability and Model Reuse. In *Proceedings of the International Conference on Modeling and Simulation*, pages 230–237, Phuket, Thailand, 2007.
- [28] A. Tolk. What Comes After the Semantic Web - PADS Implications for the Dynamic Web. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 55–62, Singapore, 2006.
- [29] A. Verbraeck. Component-based Distributed Simulations. The Way Forward? In *Proceedings of 18th Workshop on Parallel and Distributed Simulation*, pages 141–148, Kufstein, Austria, 2004.