COMPOSABLE SIMULATION MODELS AND THEIR FORMAL VALIDATION

CLAUDIA SZABO

B. Eng., "POLITEHNICA" UNIVERSITY OF BUCHAREST, ROMANIA

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2010

Abstract

In component-based modeling and simulation, shared models are reused and assembled in various combinations to meet different user requirements, resulting in an appealing approach to reduce the time and cost of developing complex simulations. Towards achieving this goal, there are several challenges including the lack of methodologies and techniques to support the life-cycle of component-based development, and the validation of semantic composability among others. This thesis focuses on two main directions: a new component-based modeling and simulation approach, and a novel semantic composability validation strategy.

The key to our proposed integrated component-based approach is the abstraction and specification of components as *meta-components* using semantically sugared attribute values from a new component-based simulation ontology. This ontology is designed to capture component behaviors and to support model composition across applications, as well as the verification and validation of the composed model. Using a *component-connector paradigm*, components in a composed model interact only through well-defined connectors. This allows us to formally specify the composition using EBNF grammars. The representation of the composed models as production strings simplifies and facilitates automated syntactic verification and model discovery and selection. Another key design decision in our approach is the modeling of real-world basic entities as *base components*, and reusable models as *model components*, to achieve greater reuse within and across application domains.

In semantic composability, our main design considerations and trade-offs are *validation accuracy* and *computational cost*. Two key observations are: in model validation, the interactions of component behaviors grow rapidly and lead to the problem of exponential state-space explosion; and current techniques that focus on checking that a model is semantically valid are computationally more costly than checking for model invalidity. Based on these two insights, we propose a new *deny validity semantic validation strategy*. Firstly, the model is validated for general model properties such as safety and liveness, which is less costly for identifying invalid models. Models that pass this test have increased credibility, and are then subjected to a more rigorous but expensive formal semantic validation. A new time-based semantic validation technique is proposed. Since semantic validity is not a fixed-point answer, we introduce a new metric to quantify semantic similarity among different models.

Our approach is validated using both theoretical and experimental analysis. We illustrate our component-based approach using three examples: a simple queueing system to introduce key concepts, a more complex component-based grid system to illustrate model component reuse, and a data-driven military training simulation scenario to demonstrate composition and validation in a new and more complex application domain. Theoretical and experimental analysis using our prototype implementation demonstrates that our approach is appealing as initially envisaged, and our deny validity semantic validation strategy provides a framework to advance the understanding of the trade-offs between validation accuracy and computational cost. Lastly, we highlight a number of new insights and research challenges.

List of Publications

1. C. Szabo and Y. M. Teo, *On Validation of Semantic Composability in Data-driven Simulation*, Proceedings of the 24^{th} ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation, pp. 73-80, IEEE Computer Society Press, Atlanta, USA, May 19-21, 2010.

2. C. Szabo and Y. M. Teo, *A Time-based Formalism for the Validation of Semantic Composability*, Proceedings of the Winter Simulation Conference, pp. 1411-1422, IEEE Computer Society Press, Austin, USA, December 12-16, 2009 (**ACM SIGSIM Best Student Paper Award**).

3. C. Szabo and Y.M. Teo, *An Approach for Validation of Semantic Composability in Simulation Models*, Proceedings of the 23^{rd} ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation, pp. 3-10, IEEE Computer Society Press, New York, USA, June 22-25, 2009.

4. Y. M. Teo and C. Szabo, *CoDES: An Integrated Approach to Composable Modeling and Simulation*, Proceedings of the 41^{st} Annual Simulation Symposium, pp. 103-110, IEEE Computer Society Press, Ottawa, Canada, Apr 13-16, 2008.

5. C. Szabo and Y. M. Teo, *On Syntactic Composability and Model Reuse*, Proceedings of the International Conference on Modeling and Simulation, pp. 230-236, IEEE Computer Society Press, Phuket, Thailand, March 2007 (invited paper).

Acknowledgements

It is with unbounded gratitude that I write these lines. I would like to thank my supervisor, Professor Yong Meng Teo, for supporting me throughout this thesis with patience and knowledge, while at the same time allowing me room to follow my own way. Professor Teo has taught me how to look at a problem, what important questions to ask, and how to present my ideas clearly, focusing on the big picture and the important insights, rather than the intricate details of methods. More importantly, through his never-ending patience and kindness in dealing with his students, he has shown me that teaching is an act of patience and understanding, rather than a simple information exchange. I am happy he was my supervisor throughout these years and could not wish for a better advisor.

I would like to thank my thesis committee, Professors Gary Tan and Stanislaw Jarzabek, for their continuous feedback and encouragement. They have both given me their precious time and provided me the resources I needed. I would further like to thank Professor Rassul Ayani and Dr Simon See for the helpful and insightful discussions on simulation, composability, and distributed computing. Dr Verdi March has always offered feedback and highlighted important issues during our seminar talks.

In the daily work in the Computer Systems Research Laboratory, I was surrounded by many great friends and brilliant colleagues. To Marian, for being a constant support throughout all the stressful times, putting up with me and making me laugh or focus when needed, there are no words to express my gratitude. To Mihai and Mariuca, thank you for all the coffee breaks and companionship throughout the first part of this quest. To Bogdan and Cristina, thank you for all the insightful conversations on life and computer science, over the never-changing food from the Science canteen. To Marius, I am grateful for showing me the power of believing in my dreams. To Sandra and Doris, thank you for being my climbing guinea pigs and sharing the cost of durians with me, although I always ate the bigger part; you've made this foreign country seem like home. To Cristina, Nandini, Dmitry, and Aleks, thanks for all the fun in this last stretch.

Finally, I thank my mother and my grandmother for raising me, supporting me through my education, and always pushing me to give my best.

Table of Contents

List of Acronyms

BOM:	Base Object Model
CADP:	Construction and Analysis of Distributed Processes
CCA:	Common Component Architecture
CoDES:	Composable Discrete-Event scalable Simulation
COM:	Component Object Model
COML:	COmponent Modeling Language
CORBA:	Common Object Request Broker Architecture
COSMO:	COmponent Simulation and Modeling Ontology
COST:	Component-Oriented Simulation Toolkit
DEVS:	Discrete Event System Specification
EJB:	Enterprise Java Beans
EBNF:	Extended Backus-Naur Form
HLA:	High Level Architecture
LTS:	Labeled Transition System
OSA:	Open Simulation Architecture

List of Tables

List of Figures

List of Definitions

List of Equations

Chapter 1

Introduction

1.1 Motivation

Modeling and simulation (M&S), the third pillar of science, is widely used in science, engineering, military training, healthcare, and manufacturing, among others, to make crucial policy decisions and answer "what-if" scenarios [11]. The fundamental building block of M&S is the *model*, an abstraction of the real system that is executed over time with different inputs in a *simulation*. Simulations provide the means to analyze complex systems without physical deployment that can be costly or even dangerous [11]. However, modeling and simulation is a costly process itself [30]. This is because simulation models can be large, monolithic artifacts that require expertise and time to develop and validate. Validation is of paramount importance, especially when models are employed for critical decisions such as in military exercises or in the evaluation of financial decisions [47, 51, 72]. For example, the Verification, Validation and Accreditation (VV&A) process for modeling and simulation, used in the US Department of the Navy, defines seven user roles and thirteen important steps grouped into five categories, and is a lengthy validation process involving many departments [35].

Since more often than not simulation models abstract real-life systems of intricate detail, the size and complexity of simulation models is hard to grasp. For example, the

core ModSAF simulator contains millions of lines of code (LOC), and took around 250 man-days for development only, without verification or validation [74, 131]. It would be an incredible loss of brain and money power if every time a large simulator is needed, its developers would start by writing everything from scratch. This is where simulation model reuse comes into play, because the reuse of simulation artifacts holds promise to drastically reduce the development time and cost, and to increase knowledge sharing to a wider user community. A quick peek at the related field of component-based software engineering shows that large components, around half a million LOC each, are easily composed in eight man-days [46]. An approach to reduce the initial costs of simulation development is to *reuse* previously developed and validated simulation components, and to *compose* them in a new simulation model according to the desired user objectives [61]. This approach, called *component-based simulation model development*, is increasingly of interest for developing complex simulations [30, 43, 61, 91].

Simulation composability is defined as "the capability to select and assemble simulation components in various combinations to satisfy user requirements" [88]. Accordingly, models developed using off-the-shelf components in an integrated component-based framework are appealing [67]. Additionally, 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, integrated component-oriented simulation framework can advance the knowledge sharing of model components to a wider simulation community.

Bartholet et al. [12] categorize composition into two main levels, namely *syntactic composability* and *semantic composability*. In syntactic composability, the composition consists of components that are properly connected and must interoperate, i.e., assume common communication protocols, data formats, as well as a common understanding of the time management mechanisms employed. In semantic composability, the com-

posed model produces meaningful behavior that meets user objectives. Furthermore, the composition execution must be exact or close enough to the real system or its representation [88]. This is because simulation models are widely used to make critical decisions [11]. As such, semantically valid and *credible* simulation models are necessary [126], and a component-based simulation framework must facilitate and validate semantic composability [61].

From the perspective of the usability of a component-based simulation framework, Kasputis and Ng [61] envisaged an integrated component-based simulation system in which requirements are specified by the user. The system builds the simulation in real time from a library of simulation models that can be easily combined to produce the desired functionality. Challenges identified in simulation composability include the specification of user requirements, module identification based on user requirements, the organization of a simulation model repository, as well as the selection of the best simulation component and the validation and verification of the composed model. Davis et al. [31] identify other issues such as component representation and discovery, the development of a valid marketplace where simulation components are bought and sold, as well as the need to decide between general simulation frameworks versus application domain-oriented frameworks.

Several component-based simulation frameworks for *specific* application domains have been developed such as in electronics [33], thermofluid [102], mechatronics [25], and computer network systems [106]. 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 key challenge is the development of a component-based simulation framework that achieves coverage both in *breadth* (across many application domains) and in *depth* (within a specific application domain) [12, 30, 85, 105], and at the same time allows for the ver-

ification and validation of component assumptions and constraints [113]. Another key challenge is to achieve syntactically and semantically valid compositions from heterogeneous components [30, 87]. Besides achieving syntactic and semantic composability, a component-based simulation framework must also support component discovery [31, 91, 128], reuse of developed simulation models [12, 31, 61, 89, 94], and integration with off-the-shelf simulation components. Component reuse is of paramount importance for a component-based framework to be accepted by the simulation community [31, 43, 61, 80, 91, 128]. In modeling and simulation, component discovery is required to locate shared simulation components stored at different administrative domains. To be scalable, component discovery must be achieved in a distributed context, where component providers and consumers reside in different administrative domains [31, 43, 61, 80]. Furthermore, the optimal selection of components based on composition objectives or purposes was shown to be an NP complete problem [13, 43, 61, 85, 94, 128].

Component-based simulations are inherently software artifacts and the simulation community can benefit from the advantages of component-based software engineering, namely, reduced development time, increased reliability, and broader acceptance. From this perspective data encapsulation, loose coupling, communication protocols etc., may be of relevance to component-based simulations [12]. On the other hand, the simulation community has argued that simulations developed from components cannot be considered simply as ordinary component-based software systems because of their complex, time-based dynamic structure [31, 61], which is more susceptible to emergent properties through composition [85]. To understand the differences and similarities, we compare and contrast component-based simulations with component-based software engineering focusing on a few important issues and using a selection of component models.

Component models can be distinguished based on three key criteria, namely, *compo-*

nent representation, *composition*, and *framework*. In *component representation*, components are classified [123] as *black-box*, where the component provides no details of its implementation beyond its interface and specification, *white-box*, where a third party has access and can modify the component's implementation, *glass-box* where the component's implementation is available for study but closed for modifications, and *gray-box* where only a controlled part of the implementation is open for third party modification. The latter type of components is considered confusing and is rarely used [123]. From a simulation perspective, we consider whether components are stateful and whether the concept of time can be clearly captured in the component representation. Statefulness and the changes of component behavior over time are two attributes of paramount importance in simulation models. The component behavior, i.e., how a component acts both independently and within a composition, has a great impact on the composed model and must be properly defined.

In a *composition*, the reuse of developed software components is twofold. The straightforward form of reuse and the most widely accepted in industry is the reuse of the standalone component in *flat compositions*. Typically, components are sold by their developers in online marketplaces such as ComponentSource [27] or Flashline [40]. Customers search for and purchase components from these marketplaces and integrate them in their software program where the purchased components interoperate with other components. In contrast, in *hierarchical composition*, components are also sub-components that are composed and subsequently reused as a single, larger component. The reuse of such hierarchical components (hereafter called *model components*) further increases the size and diversity of the component repository. Next, we analyze if components can be composed through the use of connectors, in a *component-connector paradigm*, which is a fundamental notion in Architecture Description Languages, where connectors support component interaction by providing functionality such as message

delivery, synchronization, and encryption [56]. We discuss the distribution of components over the network, an important issue nowadays when the Internet is the center point of pervasive sharing of resources of all types. *Frameworks* can be general purpose or tailored for a specific application domain. From a practical perspective, a framework must also provide a component marketplace where components can be shared or acquired for reuse.

1.2 Composability in Modeling and Simulation

This section provides a brief overview of component-based modeling and simulation (CBMS). A more in depth analysis is presented in Chapter 2. Different types of composability can be achieved among simulation components. Tolk [126] proposes a six-level taxonomy of simulation component composability, namely technical, syntactic, semantic, pragmatic, dynamic, and conceptual. In technical composability, components exchange data using a common protocol for communication. Syntactic composability occurs when components have a common data format that is unambiguously defined. In semantic composability, the meaning of data is known to all communicating components. In pragmatic composability, the use of data by each participating component is known to the rest of the components in the composition. Dynamic composability enhances pragmatic composability by requiring components to be fully aware of the state changes in the entire composition during the simulation execution. Lastly, in conceptual composability, simulation components are composable even at a very abstract level. While the taxonomy provides precious insight into the mechanics of composability, its major drawback is the fact that important considerations such as *context* and *conceptual model* are vaguely defined. We adopt an older taxonomy, which considers two levels of composability: *syntactic* (or engineering) and *semantic* [88]. *Syntactic composability* refers to component connection and interoperability and assumes com-

mon communication protocols, data formats, as well as a common understanding of the time management mechanisms employed. In *semantic composability*, the composition produces meaningful behavior according to user objectives. Furthermore, the composition execution must be exact or close enough to the real system or its representation [88]. This level encapsulates the levels in the Tolk taxonomy and captures distinctions such as meaningful component behavior in the composition, composition context, etc., which are defined in Chapter 6.

The fundamental building block of a component-based simulation is the simulation component. A simulation component is defined as [30]:

> *. . . a self-contained unit that is independently testable and usable in a variety of contexts. It interacts with its environment only through a well defined interface of inputs and outputs.*

It is important to note that the above definition assumes that each component is an independent entity that can be used meaningfully without understanding how it works. This is true for well-defined and understood components such as a random number generator or a well-known mathematical function. However, simulation components generally model specific entities with characteristic behaviors that are abstracted beyond that of input/output transformations. More importantly, the behavior of these entities varies with time and is guided by tacit assumptions and rules [113]. We present a more accurate component definition in Chapter 3.

With respect to these semantic concerns, simulation composability is [30]:

> *. . . the capability to select and assemble components in various combinations to satisfy specific user requirements meaningfully. A defining characteristic of composability is the ability to combine and recombine components into different systems for different purposes.*

An important point to highlight is that in addition to composing and recomposing components in various ways, the entire composed simulation must also meet the objectives of the simulation developers and the composition must be valid.

A number of component-based simulation frameworks have been proposed. These include application-domain oriented frameworks such as electronic systems [33], thermofluid systems [102], mechatronics [25], network [106] and military training [38] simulations. Among the general-purpose simulation frameworks, we selected three representative frameworks for discussion in this thesis, namely, the Discrete Event System Specification (DEVS) [134], Open Simulation Architecture (OSA) [29], and Base Object Model (BOM) [50]. Additionally, we discuss the formal theory of semantic composability proposed by Petty and Weisel [88].

DEVS [132] is a formalism derived from general system theory and is used to describe the structure and behavior of a system. In DEVS, a simulation model is a black-box with states, input and output ports. A DEVS model changes state whenever external or internal events occur at specified moments in time. To facilitate hierarchical composition of simulation models using the DEVS formalism, Ziegler et al. [132] introduce the concept of DEVS coupled systems, which allow for the composition of basic and other coupled systems to form larger systems. The Open Simulation Architecture [29] supports discrete-event simulation and is built on top of the ObjectWeb Consortium's Fractal [18] component model. The Fractal component model allows for the sharing of a sub-component between several distinct components, implements the separation of concerns paradigm [1], is independent of the programming language used, and supports dynamic models. A BOM is a "piece-part of a conceptual model composed of a group of interrelated elements, which can be used as a building block in the development and extension of a federation, individual federate, FOM or SOM" [50]. A BOM contains static descriptions of items resident in the real world described in terms of conceptual

entities and conceptual events. Information on how such items relate or interact with each other in the real world is expressed in terms of patterns of interplay and state machines. Petty and Weisel pioneered a formal theory for checking the semantic validity of a composed simulation model [88]. This approach includes formal representations of simulations and semantic validity that use a representation of a simulation model as a function over integer domains.

With respect to *component representation*, the four representative frameworks view a component as a black-box with a clear specification that describes its behavior. This is achieved using the DEVS formalism, a XML-based markup language, and a function over integer domains for DEVS, OSA, and Petty & Weisel's approaches respectively. BOM also employs a XML-based markup language. Components are stateful in DEVS, OSA, and BOM. The concept of time is clearly modeled in the DEVS formalism and OSA component implementations, but is missing in Petty and Weisel's theory and in BOM. The *composition* of hierarchical components is possible in all approaches. However, only DEVS, OSA, and BOM consider the reuse of hierarchical compositions. Furthermore, only DEVS adheres to the component-connector paradigm. Implementations where components are distributed over the network are possible in DEVS, OSA, and BOM. From a *framework* point of view, all proposed solutions are general-purpose frameworks. However, none of the analyzed frameworks have established any component marketplace. Table 1.1 presents a summary.

While simulation composability is appealing, several key issues still remain. Firstly, component representation is of paramount importance to achieve composability [30, 31, 61, 91, 128]. From a composition perspective, *model reuse* [12, 31, 61, 80, 89, 91, 94, 128] and the existence of heterogeneous components [12, 30, 85, 105] pose major challenges. From a process point of view, a component-based framework must facilitate *model discovery* [31, 91, 128], *syntactic composability* [32, 61, 80, 91], semantic com-

Criteria		DEVS [1997]	OSA [2006]	BOM [2004]	Petty & Weisel [2000]
Component Representation	black-box	✓	✓	✓	✓
	stateful	✓	✓	✓	-
	time	✓	✓	-	-
Composition	hierarchical components	✓	✓	✓	✓
	component-connector paradigm	✓	-	-	-
	distributed components	✓	✓	✓	-
Framework	general-purpose	✓	✓	✓	✓
	component marketplace	-	-	-	-

Table 1.1: Component-based Simulation Approaches

posability [12, 32, 88], and *model validation* [30, 32, 61, 80, 81, 89, 113] to increase the credibility of the composed model. A component-based simulation framework must also consider implementation issues such as time management [81, 91] in component-based simulations, and the development of component repositories [31, 43, 61, 80]. Furthermore, component selection is an NP complete problem [13, 43, 61, 85, 94, 128]. We analyze these complex issues in Chapter 2.

A key issue for composability is achieving and validating semantic composability, preferably through a formal process to increase credibility. However, to date, efforts towards semantic simulation composability have mostly focused on theoretical aspects, such as complexity measures of the component selection problem [85], formalizing DEVS compositions [127], and a formal theory of composability [88, 87]. There is a lack of a practical validation process of semantic composability. In contrast, simulation interoperability has seen advanced developments in the past few years, which is one of the focuses of this thesis. Efforts such as the Distributed Interactive Simulation (DIS) [55], the Aggregate Level Simulation Protocol (ALSP) [39], and the High Level Architecture (HLA) [28], have facilitated the grouping of simulations into interoperable federations. However, some significant drawbacks remain. Firstly, these efforts only ensure that simulation components can interoperate, and do not provide any guarantee on the validity of the composition. As such, there is no insight into whether the com-

bined computation of the interoperable components serves the objective required by the simulation developer. Another drawback of simulation model reuse, is that it is not possible without significant source code modifications. This is because the simulation components are not designed for reuse in a variety of context or in anticipation of different user objectives. A seamless, transparent reuse process can increase the size of the component repository and advance knowledge sharing to a wider user community.

1.3 Composability in Software Engineering

In component-based software engineering (CBSE), the development of new software products by combining shared, purchased or developed components improves software quality and supports rapid development, leading to reduced cost and time to market [123], as shown more recently by Garlan's integration study [46]. A component-based approach is even more appealing when components are used as fundamental artifacts in product line approaches, which see the development of software products similar to assembly lines in car manufacturing plants [26, 58]. Furthermore, the modular design of the software artifacts reduces maintenance costs when only individual components need to be modified or replaced. The component-based software industry has seen a major step forward in the 1990s with the introduction of component models such as the Object Management Group's CORBA (1991) [2], Microsoft's COM (1991) [97], and IBM's (and later Sun's) EJB (1997) [103], as well as academic solutions such as CCA (2006) [5], Darwin [68], and SOFA (1998) [90]. This interest has resulted in various software component marketplaces. For example ComponentSource [27] listed 1,098 software components in 2001, out of which 90% are COM and .NET components, 8% are EJB and JavaBeans components, and 2% are CORBA components [123]. However, despite the extensive industry and academic research investment, component-based engineering has not been as widely adopted as initially predicted [34]. In 2009, the num-

ber of components offered by ComponentSource increased to 1,446 components, with COM and .NET components making up 95% of the products, and EJB and JavaBeans components the other 5%.

Szyperski [123] defines a software component as:

> *... a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.*

Several component characteristics can be drawn from this definition. Firstly, a software component is a *unit of independent deployment*, well separated from its environment and other components. Secondly, a component must be *fully deployable*, which implies that it must be sufficiently self-contained, with well-defined interfaces and specifications of the component behavior and requirements. Furthermore, the component contract describes the component dependencies and interfaces, as well as how the component is deployed, instantiated, and how the component interfaces behave. We selected for discussion four component-based software engineering approaches, namely, the industry solutions CORBA [2], EJB [103], and the COM [97] component models, and the academic solution CCA [5].

The Common Object Request Broker Architecture (CORBA) is a standard proposed by the Object Management Group in 1991 to facilitate interoperability in heterogeneous computing environments [2]. A CORBA object is an accessible entity that can receive client requests for its methods. It is described in terms of data structures and method signatures using an Interface Description Language (IDL). The communication between client and server objects is done through Object Request Brokers (ORB), which is a middleware that allows the programmers to make program calls and pass object references from one computer to another via a computer network. The ORBs communicate through a well-defined protocol.

Enterprise JavaBeans, proposed by IBM and Sun in 1999 defines an architecture for the development and deployment of server-side, distributed component called enterprise beans [103]. The beans reside in containers that facilitate the provision of remote services for clients distributed throughout the network. Three types of enterprise beans are defined in the EJB specification, namely, entity, which represent data; session, which manage interactions between entity and other session beans; and message driven beans, which handle asynchronous messages. EJB provides a specification that allows for ease of switching between different EJB Container providers. Furthermore, since most of the enterprise bean management is left up to the EJB Container, the vendor of the container can transparently scale server-side resources to meet with increasing or changing demand. However, the complex EJB specification leads to high development times and often results in an overly complex solution. Furthermore, hierarchical composition between different enterprise beans cannot be achieved.

The Component Object Model (COM) introduced by Microsoft in 1999 is a binary interface standard for software components that facilitates interprocess communication and dynamic object creation [97]. In COM, a component expresses its functionality through one or more interfaces. `IUnknown`, the standard interface all COM components must implement, is responsible for reference counting, lifetime management, and access to the rest of the component interfaces. In Distributed COM (DCOM), clients and servers interact across the network, similar to the CORBA model. In the COM model, component reuse is achieved in two-fold, either by component containment (one *outer* component containing a *inner* component), or by component aggregation, in which the outer component exposes interfaces from the inner component as if they belonged to the outer component.

The Common Component Architecture (CCA) is a framework for high performance scientific computing introduced in 1999 by Sandia National Laboratories [5], focusing

towards fast connections among components that perform numerically intensive work but also towards parallel collective component interactions that use multiple processes or threads. CCA components are connected using connection ports that adhere to the *provides/uses* paradigm, to facilitate both loose coupling for maintainability and adaptability, and tight coupling for increased execution speed. In the provides/uses paradigm, one port can *use* methods that are *provided* by other ports from different components. A major advantage is that the framework decides how the provides/uses interfaces are connected, i.e., whether methods are called directly or through interface proxies. However, the focus of CCA is towards high performance *scientific* computing where most components generally perform numerical and algebraic operations. Thus, components are stateless and there is no support for time and hierarchical composition.

Table 1.2 presents a summary of the four approaches. As shown, all approaches

Criteria		**CORBA** [1991]	**CCA** [1999]	**EJB** [1999]	**COM** [1999]
Component Representation	black-box	✓	✓	✓	✓
	stateful	✓	-	-	✓
	time	-	-	-	-
Composition	hierarchical components	-	-	-	✓
	component-connector paradigm	-	✓	-	-
	distributed components	✓	✓	✓	✓
Framework	general purpose	✓	-	✓	✓
	component marketplace	✓	✓	✓	✓

Table 1.2: Component-based Software Engineering Approaches

represent components as black-boxes, but components are required to expose their public interfaces, either through reflection, stubs, or a combination of both. Moreover, there is no support for describing the behavior of a component. The published public component interface has a list of methods that the component provides, (i.e., the component syntax), but without support to describe what the listed methods actually do, (i.e., the semantics). This disadvantage is addressed in academic solutions such as Darwin

[68] and SOFA [90]. However, academic solutions focus mainly on the design of the component-based framework and generally lack an implementation [3, 68].

An important point to highlight is that component-based software engineering approaches focus mainly on the syntactic composability of components. They provide implementation facilities for components to interoperate, but without provisioning for the quality and meaning of the information exchanged between components in the context of the entire composition behavior. Lastly, when developing component-based software, most users notice small "side-effects" which lead to unexpected increases in development time. These side-effects include architectural mismatches [46] and arise because of tacit and conflicting component assumptions (data, structure, control, creation), component connections, and the general structure of the composition. On the other hand, a different study in which components were designed to work together has found none of the mismatches the previous study discovered [117]. This has led to the conclusion that "if components are to be composable, they have to be designed for it". However, this advanced level of detail is difficult to achieve in a general purpose component-based software framework.

1.4 Similarities and Differences

Similarities between component-based software engineering (CBSE) and component-based modeling and simulation (CBMS) stem from the fact that simulations are inherently software artifacts, and as such the approaches towards component representation, and composition are similar. More importantly, the concerns that govern research in both fields include scalability, usability, credibility, and a wider community reach.

However, there are several key differences between CBSE and CBMS. Firstly, in both domains components are represented independently of the underlying programming language. However, current CBSE approaches describe only the *syntax* of the

component methods to facilitate component connection and composition execution [2, 97, 103]. In contrast, the focus of CBMS approaches is to describe *component behavior* to offer a better understanding of how the component affects the simulation execution. With respect to composition, most software engineering solutions focus on achieving *syntactic composability* by providing a stable runtime environment where components are connected and executed. In contrast, semantic composability is highly desired in component-based modeling and simulations. However, efforts have focused mainly on achieving syntactic composability with success stories such as SIMNET [95], ALSP [39], HLA [28], and OneSAF [131].

A first step towards semantic composability is achieving a common understanding among components through the use of ontologies [15]. An ontology is an organized knowledge representation to capture object information in a particular domain [112], 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 [17]. In software engineering, ontologies are widely used to facilitate component discovery [66, 116, 124]. In modeling and simulation, ontologies such as DeMO [108, 109] provide a starting point towards achieving semantic composability by facilitating conversions from application domain specific knowledge to simulation concepts. To the best of our knowledge, there is a lack of an integrated framework that supports the development life-cycle of CBMS, including syntactic and semantic composability, component discovery and selection, repository management, and composition execution.

Another critical issue in composability is composition validation. In software engineering, validation focuses on the overall program correctness [3]. This includes validating logical properties such as safety, liveness, and deadlock freedom. Other approaches focus on ensuring that component methods are executed in the correct order

according to some protocols [63]. In contrast, a valid simulation model is one that mimics closely the real system that the simulation model abstracts [10]. Here, while overall program correctness is required, it is very important for the composed simulation model to produce results that are close enough to the real system that it abstracts. Very often, this similarity cannot be fully captured by an automated validation process because it depends on both input/output transformations, i.e., the simulation model must have almost the same output as the real system when presented with the same input, as well as finer points such as overall simulation model state and unified component assumptions and context [30, 126]. In contrast to software engineering systems where the verification process is highly automated, the validation process in modeling and simulation is often manual, lengthy, and requires the presence of a system expert [10, 11]. This is also because the simulation model is often used in critical situations where a valid answer is crucial. For example, the validation of military training simulations [51, 72] requires a lengthy VV&A process [35] conducted by system experts.

Table 1.3 summarizes the key differences and similarities between component-based software engineering and component-based modeling and simulation. Differences arise mainly from the stateful and dynamic nature of simulation components. Next, semantic composability is required for the composed simulation model, whereas in software engineering it is generally not considered. Furthermore, because simulations are often used to make critical decisions, a valid composed simulation model is of paramount importance. In contrast to software engineering where validity refers to program correctness, in modeling and simulation a valid composed model is one that is close enough in terms of its execution and output to the real system the composed model abstracts.

Criteria	Component-based Software Engineering	Component-based Modeling and Simulation
Component Representation	• describes method syntax [123] • stateless [5, 111, 115] • no time [111, 115, 123] • simple components [123]	• describes component behavior [132] • stateful [30, 61, 75, 88] • time based [61, 12, 132] • complex models [30]
Composition 1. Syntactic Composability	• runtime environment for component connection and composition execution [2, 5, 103]	• runtime environment for component connection and composition execution [22, 28, 29, 133]
2. Domain Knowledge	• ontology used for discovery [123]	• ontology used to translate application domain concepts to simulation concepts [109]
3. Semantic Composability	• not considered [2, 5, 103, 63]	• desired [12, 61, 88]
4. Validation	• program correctness [123, 68]	• program correctness + closeness of composed model to real system [10, 88]
Framework	• integrated framework desired [123]	• integrated framework desired [61, 12]

Table 1.3: Comparison of Component-based Software Engineering and Component-based Modeling and Simulation

1.5 Objective

Current work on component-based modeling and simulation is generally piecemeal [29] or application-specific [106], and there is a need for an integrated approach to support the life-cycle of component-based development [61]. A coherent approach can be adopted in addressing crosscutting life-cycle issues, with several benefits including among others reduced development time and cost, and increased scalability and sharing across application domains.

An important challenge in composability is the semantic validation of component-based models [30, 88, 126]. In this context, a crucial question is:

> *Given a simulation model composed using reused components, how can we systematically validate and measure (or estimate) the semantic validity of the composed model?*

The importance of semantic validity is confirmed by a recent finding of the World Technology Evaluation Centre (WTEC 2009) [48], which states that "**without validation, computational data are not credible, and hence, are useless**" [48], especially when simulation models are used to support critical decision-making [51, 72]. The definition of semantically valid models encompasses meaningful and useful behavior of the composed model with respect to the different objectives of the simulation developer. Studies of semantic composability validation reveal that the validity of a model is not a fixed point and there are many valid models, with different degrees of validity [75, 88, 121, 122]. Current approaches to validate composed models are theoretically elegant, but are still not widely-adopted [88], or are computationally expensive and thus have limited scalability [75]. The semantic validation of composed models faces many challenges from the stateful and dynamic nature of the simulation components, as well as the size and complexity of the composed models among others [30, 88, 121]. Another

development is that increasingly large simulations can be executed on Internet-based infrastructures such as Grid [64] or cloud computing [4], where higher computational capacity will be readily available.

The objective of this thesis is twofold. Firstly, we propose the design of an integrated approach that addresses the identified key crosscutting life-cycle issues in component-based simulation development. Our approach aims to support the development of more complex simulation models where simulation components can be shared within and across application domains. Secondly, we propose a strategy for the efficient validation of semantic composability that provides trade-offs between cost and credibility.

Key challenges towards our proposed goals span several areas. Firstly, towards the composability of simulation models, several crosscutting issues to be addressed are: a component abstraction that describes a simulation component from various perspectives and the representation of a composed model suitable for discovery, verification, and validation. Secondly, the cost of model validation is influenced by the degree of modeling in terms of the number of components and their attributes, and the degree of component interaction in terms of events. Devising a feasible semantic validation approach and associated techniques remains a challenge.

1.6 Contributions

The key contributions of this thesis are:

1. **Approach for Addressing Crosscutting Life-Cycle Issues** [120, 125]

   We propose to integrate solutions to crosscutting issues such as component abstractions, the representation of the composed model, knowledge representations, and model reuse, to facilitate the component-based simulation development process at low costs [125].

(a) **Meta-component Abstraction**

We propose the abstraction of a simulation component in a meta-component with semantically-sugared attribute values defined in COSMO, our component-based simulation ontology [125]. The meta-component describes a simulation component from various perspectives, both externally, with respect to neighbors in the composition, and internally, as a time-based state machine, to facilitate the composition of simulation components in a conceptual model [118, 125].

(b) **Representation of a Composed Model**

In our proposed black-box component-connector paradigm, a conceptual model is represented by a connection of black-box entities defined by meta-components. The composition adheres to several rules that are defined in an EBNF composition grammar [118]. This composition grammar specifies connection rules within and across application domains and is employed to verify the correctness of the conceptual model *before* the components are discovered in model discovery and selection [120]. In model discovery and selection, a simulation model can be internally expressed as a production string validated by the composition grammar for fast discovery of shared components. Towards meaningful ranking of partial matches, we propose to employ the semantically sugared meta-component to quantify component similarity [125].

2. **Deny Validity Strategy for Semantic Validation** [119, 121, 122]

We propose a *deny validity* approach that promises increasing levels of accuracy and credibility. Our two-step process incrementally eliminates invalid models, with increased accuracy and cost.

(a) **General Model Properties**

Our validation approach first discards invalid models through the validation

of general model properties, such as safety and liveness for instantaneous and timed transitions [119, 121]. We cover various perspectives on the definition of model properties, such as formal, practical, timeless, and timed, among others. Moreover, we propose a composability index as a measure of the degree of data alignment in the composition [125].

(b) **Time-based Formalism**

Models that have passed the first validation step might still be invalid. Furthermore, to increase model credibility, formal guarantees and measures are required. Towards a formal guarantee of the composed model validity, we perform formal validation with respect to a reference model using a novel time-based formalism [121, 122]. As a certificate of quality of the validity of the composed model, we introduce the semantic metric relation V_ϵ, which quantifies state similarities based on semantically-sugared components defined in our component-based ontology.

1.7 Thesis Organization

The outline of this thesis is presented as follows.

Chapter 2 - Related Work. We present a critical analysis of current work in component-based modeling and simulation. We establish the burning issues in simulation composability and evaluate current work. Our findings are that current challenges of component-based modeling and simulations include, among many, the reuse of simulation models towards large, infinitely scalable compositions, the lack of component representations to facilitate reuse and automated validation, meaningful model discovery based on a user query, automated syntactic composability verification, providing for semantic composability, and the validation of semantic composability.

Chapter 3 - Proposed Approach. We present an overview of the problem domain and our strategy. We identify key crosscutting issues in composability and propose an approach to address these issues in the life-cycle of a component-based simulation model. Our focus is not only on solving these individual issues, but also towards a component-based approach that integrates each life-cycle step solution in an efficient process. We introduce the preliminaries of component representation and organization, and domain knowledge representation in our proposed component-based simulation ontology called COSMO. Our proposed component representation abstracts a component as a *meta-component* with behavior and attributes. We organize components into base components, which are fundamental entities specific to application domains, and model components, which are composed using base and other model components. Lastly, we propose a four-step life-cycle in component-based model development: conceptual model definition, syntactic composability verification, model discovery and selection, and semantic composability validation. In contrast with current approaches, syntactic composability is verified *before* model discovery.

Chapter 4 - Model Composition and Verification. In our proposed framework, a simulation problem is translated into a conceptual model using a graphical environment by drag-and-drop icons representing conceptual components and framework connectors. We present our approach towards the syntactic verification of the conceptual model. In contrast to current approaches, syntactic composability verification is performed *before* the costly model discovery and selection. We describe how this is possible and present our approach to the formalization of syntactic composability through the use of compositional grammars.

Chapter 5 - Model Discovery and Selection. With a large repository of simulation

components from different sources, automated discovery and selection based on a user query are difficult without means of ranking the components meaningfully. We present a semantic measure of the similarity between query and repository components using our proposed component-based ontology. We study the feasibility of our approach on a component repository with up to two thousand simulation components.

Chapter 6 - Semantic Composability Validation. Our study of semantic composability validation shows that in the literature there are various degrees of model validity and validity is not a yes/no answer. Current approaches to validate composed models are either theoretically elegant but not implementable, or are computationally expensive and do not scale. Based on our composability studies, we observe that there are more invalid than valid models. As such, checking for invalid models is less costly on average if we employ a dual-step *deny validity* strategy. Firstly, invalid models are eliminated by validating general model properties for both instantaneous and timed transitions. If the model passes this test, formal model execution validation can then be performed. We divide the validation of semantic composability in two layers, namely the *validation of general model properties*, and the *formal validation of model execution*. We present the first layer in which desired model properties such as safety and liveness are evaluated in the context of instantaneous transitions and over time.

Chapter 7 - Formal Validation of Semantic Composability. We present the second layer of the semantic composability validation process. We propose a new time-based formalism to facilitate the comparison between the composed model and a reference perfect model. In contrast to current approaches, in which components are represented statically as functions over integer domains, we represent a component as a function of states over time. We formalize composition, simulation and validity. Based on these

formal definitions, we propose a five-step validation process. To quantify validity, we propose a semantic similarity metric to evaluate the closeness of the composed model to a reference model. We present a theoretical and experimental analysis of the validation process.

Chapter 8 - Prototype and Evaluation. In this chapter, we present the implementation of our proposed approach and evaluate each life-cycle step theoretically and through experimental analysis. Throughout this thesis, we have used a component-based single server queue system as an example. In this chapter, we follow the life-cycle of a more complex grid simulation system, which is composed using reused base and model components. We further show how a new application domain, Military Training Systems, is added to our proposed CoDES (Composable Discrete-Event scalable Simulation) framework. Components in the Military Training application domain are data-driven, complex entities, and the validation of models in this application domain requires a trade-off between computational cost and validation accuracy. In terms of increased scale, we evaluate the runtime cost of each life-cycle step for models composed from up to 1,000 components. Our results show the feasibility of our approach but also highlight new research challenges.

Chapter 9 - Conclusion. We present a summary of the key contributions of this thesis and discuss directions of future work, including the reuse of validated model components and the evaluation of emergent properties in the composed model.

Chapter 2

Related Work

Although simulation composability is an appealing approach to reduce the time and cost of developing simulation applications, several issues prevent it from becoming the silver bullet to modeling and simulation as initially envisaged [12, 61]. These issues and the state-of-the-art approaches to component-based simulation model development are discussed in detail in this chapter.

The component-based simulation framework landscape includes specific application domain approaches, such as electronic systems [33], mechatronics [25], and computer network [106] simulations, and general-purpose frameworks such as the Discrete Event System Specification (DEVS) [133], the Open Simulation Architecture (OSA) [29], the Base Object Model (BOM) [50], and the Component Oriented Simulation Toolkit (COST) [22]. However, to the best of our knowledge, current frameworks address only partially the key aspects of the component-based simulation life-cycle, such as model discovery and selection, syntactic composability verification, and semantic composability validation.

Generally, composability is achieved by ad-hoc development and connection of components in a simulation model that can only be reused "as-is" [29]. This is in contrast to an ideal setting, where a simulator developed from reusable components

can be further reused as a simulation component in various contexts and combinations [61]. Furthermore, important aspects such as the validation of syntactic and semantic composability are left for the simulation developer [12, 29, 133]. It can be said that current approaches fail to offer an integrated framework for component-based simulation model development. Such an integrated framework for composability is not merely a collection of procedures and processes that address the above life-cycle, but rather a well-oiled ensemble of parts designed to work together to achieve maximum efficiency at minimum costs [61]. On the other hand, several approaches target singular issues such as model discovery and reuse [24, 108], and semantic composability validation [88]. This piecewise approach has the potential to achieve significant breakthrough in the specific issue, but generally fails because of the lack of support from an integrated framework. For example, a formal theory of composability [88] relies on an integer function representation of a simulation component, which does not capture real simulation components.

As discussed in the previous chapter, key issues in component-based modeling and simulation include *model reuse* [12, 31, 61, 80, 89, 91, 94, 128], *component representation* [30, 31, 61, 91, 128], *model discovery* [31, 91, 128], *syntactic composability* [32, 61, 80, 91], semantic composability [12, 32, 88], and *model validation* [30, 32, 61, 80, 81, 89, 113]. Furthermore, component selection is an NP-Complete problem [13, 43, 61, 85, 94, 128]. With respect to the execution of the composed model, several challenges are posed by the heterogeneous nature of the components [12, 30, 85, 105] with respect to geographical location and different time management algorithms [81, 91]. Lastly, a component repository in which components are stored after being developed must be in place [31, 43, 61, 80].

We analyze current status of component-based simulation model development in two parts. Firstly, we present state-of-the-art approaches to component-based modeling

and simulation, including the Discrete Event System Specification (DEVS) [132], the Open Simulation Architecture (OSA) [29], and the Base Object Model (BOM) [50]. Next, we discuss key issues of component-based modeling and simulation, and how they are addressed in current approaches.

2.1 Current Approaches

2.1.1 Frameworks for Composable Simulations

Discrete Event System Specification

The Discrete Event System Specification (DEVS) [132] is a formalism derived from general system theory that describes the structure and behavior of a system. Work that embraces the DEVS formalism for component-based modeling and simulation shows that the formalism alone is not sufficient for a complete approach to component-based modeling, composition, and validation [24, 127, 133].

In DEVS, a system is modeled as a black-box with states, input and output ports. A DEVS model changes state whenever external or internal events occur at specified time moments. Two types of DEVS models exist, namely, *atomic* and *coupled*. An atomic DEVS is a tuple

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, ta, \lambda \rangle$, where

X is the set of input values

S is the set of states

Y is the set of output values

$\delta_{int} : S \rightarrow S$ is the internal transition function

$\delta_{ext} : Q \times X \rightarrow S$ is the external transition function, where

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set

e is the time elapsed since the last transition

λ : $S \rightarrow Y$ is the output function

ta : $S \rightarrow \mathbf{R}^{+}_{0,\infty}$ is the time advance function, where 0 means that the stay in state s is so short that no external events can happen (s is thus a transitional state), and ∞ represents the state s in which the system remains until an external event occurs.

The DEVS coupled models facilitate hierarchical composition [132] by allowing for the composition of basic and other coupled systems to form larger systems. Figure 2.1 shows a DEVS coupled model with two components.

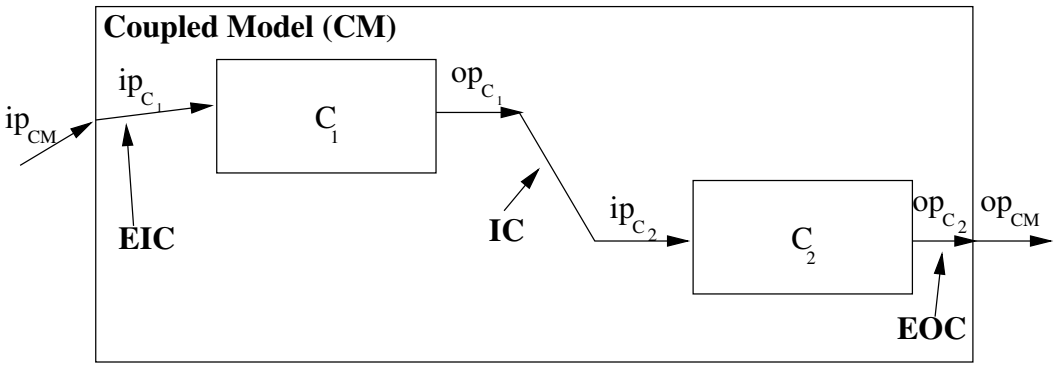


Figure 2.1: A DEVS Coupled Model

Recent work on DEVS for component-based modeling and simulation includes the implementation of DEVSJava [133], a framework for the development of DEVS component - based simulations which provides base classes for component development as well as utility libraries that contain queues, stacks, random number generators, and distributions. The Java base classes for component development correspond to the DEVS types of models, namely *atomic* and *coupled*, but also contain utility classes for properly defining coupled models as well as message passing between components. To the best of our knowledge, syntactic composability is not checked in DEVSJava. However, syntactic composability could be verified if it is assumed that the DEVSJava components correctly implement the DEVS formalism. Furthermore, model discovery and selection,

as well as semantic composability validation are not addressed. While the DEVSJava framework could be enhanced to support the above life-cycle steps, meaningful model discovery and selection as well as efficient semantic composability validation would not be possible by employing only the DEVS formalism. This is because the framework would require a component abstraction on top of the DEVS formalism to capture semantic knowledge about the framework components. Additionally, knowledge representation to facilitate reasoning about similarity and validity is needed.

Open Simulation Architecture

The Open Simulation Architecture (OSA) [29] is built on top of the ObjectWeb Consortium's Fractal component model to support component-based discrete-event simulations. While extensive work has been done towards the OSA implementation, its design does not consider abstractions to facilitate model discovery and semantic composability validation, making it difficult to extend to provide for a complete modeling, composition, and validation process.

The Fractal component model presents some appealing features. To facilitate component reuse, it allows for the sharing of a sub-component between several distinct components, and implements the separation of concerns paradigm [1]. Furthermore, the Fractal component model is independent of the programming language that is used to program simulation models and components. In the Fractal specification, a component is a unit of object-oriented code with server and client external interfaces. Components may have a hierarchical structure. Hierarchical components consist of a controller part (also called a *membrane*), and a content part, which can contain one or more components. Since a membrane can contain another component that in turn has an external interface, it follows that hierarchical components have internal and external interfaces, both of either type client or server. The internal interfaces are only available to components from the content part. A component of the inner part can only bind its interfaces

to the external interfaces of other inner components (normal binding), or to the internal interfaces of the surrounding membrane (export and import binding), as shown in Figure 2.2.

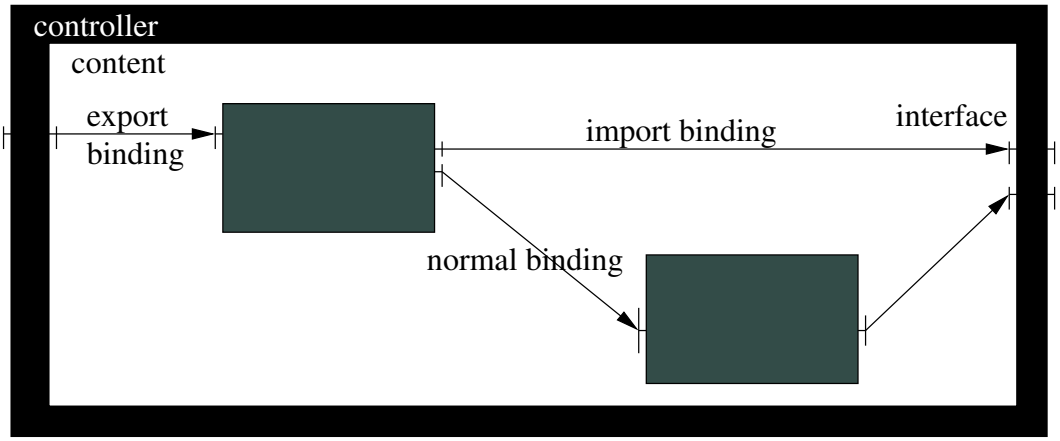


Figure 2.2: A Fractal Hierarchical Component

Similar to the DEVSJava framework, syntactic composability and verification may be achieved in the OSA framework by the strict implementation of the Fractal model, but verification is not currently considered to the best of our knowledge. Moreover, model discovery and selection, as well as semantic composability validation are not addressed. Another important point to highlight here is that in OSA there is no high-level representation of an OSA component, with OSA components represented strictly by their code. For semantic composability validation to be possible, a meta-component representation that factors in behavior and meaningful description of attributes is necessary.

Base Object Model

The Base Object Model (BOM) is "a component-based standard describing reusable piece parts of a simulation or simulation space" [50]. Initially intended as a standard

description of simulation components in an HLA setting and later adapted for general component-based simulation, it provides a general component abstraction to facilitate composability. However, the BOM description does not consider time, an attribute of paramount importance in simulation. Furthermore, there is a lack of adoption in an implemented framework that provides a complete composition life-cycle from the conceptual model to a valid composed model.

Figure 2.3 presents the structure of a BOM. The BOM Model Identification serves

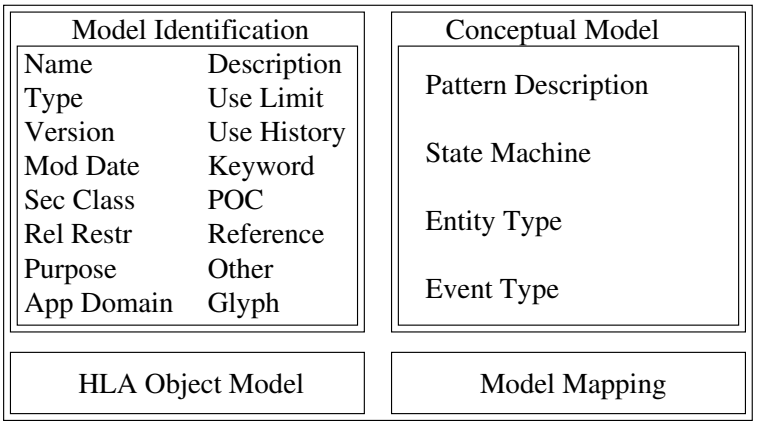


Figure 2.3: BOM Structure

to provide general information about the base object model, such as name, author, etc. The BOM Conceptual Model contains static descriptions of items resident in the real world that are described in terms of conceptual entities and events. Information on how such items relate or interact with each other in the real world is expressed in terms of patterns of interplay and state machines.

Current work that employs BOM [69, 77] addresses syntactic composability verification, model discovery, and semantic composability validation [1]. Here, components are discovered based on a detailed user-specified simulation scenario that includes event

[1] As of late 2010, a new BOM Experimental Schema proposes the composability of BOM events. However, the applicability of this new proposal for component-based simulation modeling remains to be studied.

names and parameters. In BOM Matching and Composition [77], syntactic discovery (based on event name and parameters) as well as semantic discovery (based on entity and data types according to an ontology) is performed. After discovery, candidate components are composed and their composition is executed. A composition of candidate components is valid if its execution conforms to the scenario. As such, composition and semantic validation of the composability of BOM components is done based on well-specified simulation scenarios with low level details which are costly to define in practice. Furthermore, the discovery-composition-validation cycle is computationally expensive because all possible combinations of candidate components are composed and executed.

2.1.2 Validation of Semantic Composability

Theory of Composability

Petty and Weisel pioneered a formal theory of composability for checking the semantic validity of a composed simulation model [88] using a static representation. As such, the approach cannot be applied to real-life simulations with complex connections (i.e., fork and join) and where time is an attribute of paramount importance.

A composition is modeled as a mathematical functional composition. A simulation is a sequence of executions of a model f, where the state of the previous execution iteration m_x is always fed into the next iteration together with some input. Each iteration produces an output o_x, as illustrated in Figure 2.4 (adapted from [88]). All inputs, outputs and memory values are integer values.

iagrams/simulation.pdf

Figure 2.4: Simulation as a Sequence of Executions of a Model

The simulation of a composition is represented as an Labeled Transition System

(LTS) [114] where nodes are model states, edges are function executions, and labels are model inputs. A composition is valid if and only if its simulation is close to the simulation of a perfect model, depicted by M^* in Figure 2.5.

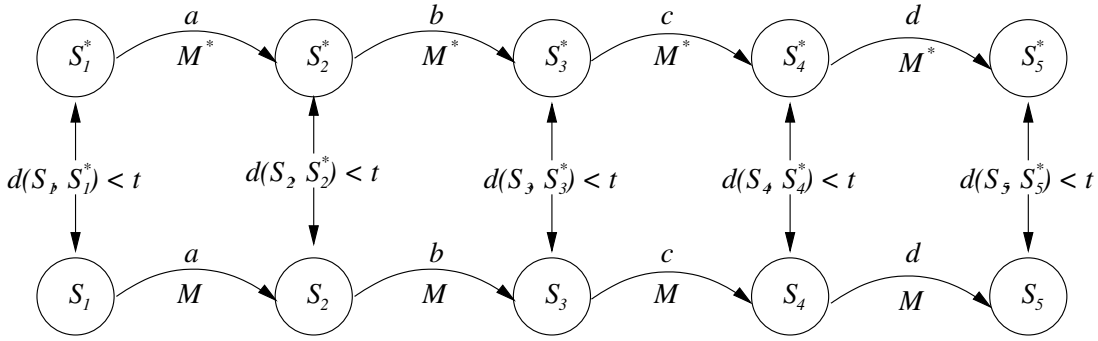


Figure 2.5: Composability Validation by Comparison with Perfect Model

In this approach, time is not modeled and the function representing a component assumes an instantaneous transition from input to output. This permits only a *static* representation of the composition that does not consider *time*, an attribute that characterizes most simulation components. Furthermore, the LTS representation considers the functions strictly in the order they appear in the mathematical composition, which is not representative for compositions with complex structure such as those with fork and join connectors.

2.2 Major Design Issues

Key issues in component-based modeling and simulation include *model reuse*, *model discovery*, *syntactic composability*, *component representation*, semantic composability, and *model validation*. For the execution of the composed model, several challenges are posed by the heterogeneous nature of the components with respect to geographical location and different time management algorithms. In addition, a component repository in which shared components are stored after being developed must be in place.

2.2.1 Model Discovery and Reuse

Reuse is defined in software engineering as "the process of building or assembling software applications and systems from previously assembled parts designed for reuse" [70]. The benefits of reuse [44, 80] include shorter development time, as well as lower maintenance costs and better product quality. A simulation model can be reused as a standalone simulator or as a model component interoperating with other model components in a larger simulation model. A reuse process includes methodologies, techniques and procedures for defining, selecting, and composing simulation components [89, 94], as well as for saving the composed model into the component repository. Reuse solutions in software engineering have not proved as successful as envisaged [78], mainly due to the lack of procedures and techniques for building, maintaining, and querying a component repository. Another contributing factor is the time-costly component integration process, which ironically is the exact effect that component reuse aims to remove. Similarly, in modeling and simulation challenges lie in building reusable simulation models [61], integrating legacy systems in a meaningful manner [109], and formally defining scenarios for reuse [89].

A DEVS model can be employed as a standalone simulator, but also as part of larger DEVS models. The latter is achieved through the coupled DEVS formalism that includes atomic DEVS models as well as other coupled DEVS models. Component reuse is achieved in OSA through the implementation of the Fractal component model, which allows for a component to have sub-components. In frameworks that employ the Base Object Model [75], BOMs can be reused as they are, as individual entities, but cannot be part of larger BOMs because the BOM is not hierarchical in this way by definition. The Petty and Weisel approach does not consider model reuse.

While current approaches such as DEVS and OSA facilitate the reuse of developed components, there is currently a lack of support for the reuse process. As discussed

above, model reuse is an integrated process, which includes the discovery of reusable models, and the execution of the composed model. Meaningful model discovery is difficult to achieve in DEVS because the DEVS formalism does not fare well when an ontology is attached. This is because of the functional definition of a DEVS component. Similarly, in OSA, the component representation focuses on the component implementation, which does not facilitate meaningful discovery.

Discovery refers to the process of locating a component from a component repository with a specific structure and component representation [31, 61, 91]. With respect to the way in which the user specifies his query, several types of discovery can be identified. Firstly, if the user specifies a set of objectives that the composition should meet, discovery of the *entire composition* is performed. This process will attempt to discover a set of components, which meet some objectives themselves, whose composition meets the desired user objective. However, the composition might result in some unexpected or emergent behavior. This makes this type of discovery an NP-Complete problem [13, 85]. Secondly, the user might employ some form of modeling of the composed model, such as drawing or building the composition using drag-and-drop of icons on a drawing panel [120]. Next, the user specifies individual queries for each component. In this case, the query focuses on the individual component characteristics and the composed model must be validated once the discovery of all the individual components has finished. Key issues in component discovery include the response time of the discovery service, which mandates that components be discovered quickly. Another important factor that influences the performance of the discovery service is the size of the component repository, because discovery is based on a comparison between the query component and each component in the repository in the worst case. Furthermore, it is rarely the case that exact matches to the query exist. In this case, criteria to identify and select partial matches are needed. Other factors include the representation of compo-

nents in the repository [61, 91], and the existence of a component marketplace [31, 80] in which components are traded.

The DEVS and Petty and Weisel approaches do not consider component or model discovery. Moreover, meaningful discovery is not possible in Petty and Weisel's approach because of the functional component representation that considers only integer values with no semantic meaning. The OSA component representation allows for the specification of the name of a dependency component that can be discovered. This is a very basic form of discovery that cannot be extended to the process described here. Current work that employs the BOM representation [75] uses a low level scenario to describe the desired composed simulation. The scenario includes the sequence of component execution, as well as events and parameter names for interacting components. Component discovery is performed based on the specified scenario, considering the syntax (method parameter) and the semantics (sequence of events) of the components. A valid composition of discovered components is one in which the sequence of actions or events is the same as the sequence specified in the scenario. This requires that all possible compositions of components (since there can be more than one candidate for a specific component) be executed and compared with the user specified scenario. However, this is time consuming and computationally expensive. Furthermore, the approach considers only exact matches.

2.2.2 Component Representation

Component representation [31, 61] refers to the description or abstraction of simulation components. To facilitate composability, the component representation should contain information about the component behavior, its connection with other components in the composition, as well as input and output data. A component representation abstracts the component implementation to a level that is understandable by all involved in the

composition process, humans and computers alike. The general view in modeling and simulation is that a standardized *meta-description* or a *metamodel* should accompany the component implementation, but what it should contain is still subject to debate [61]. This is because the component representation is used in model discovery and selection, in syntactic composability verification, and in semantic composability validation, which implies a trade-off between the required levels of detail for each of these steps [30, 31, 61, 91]. Furthermore, the existence of different application domains may result in various types of behavior information that needs to be captured [30].

The DEVS formalism represents a component through internal and external functions that change state following a well-defined time-advance function. BOM is a meta-representation in itself covering model description and behavior. However, the representation does not consider time, an attribute of paramount importance in simulation. Petty and Weisel represent a component as a mathematical function over integer domains. This representation simplifies the validation of composed models by reducing it to functional mathematic composition. However, other parts of the composability process, such as meaningful model discovery and syntactic composability verification are not possible and as such not considered by the approach. Furthermore, the Petty and Weisel representation cannot be used to describe complex real-life components because varied component attributes and behaviors are difficult to match to single-coordinate domains. The OSA component representation focuses only on implementation specific details such as the component location and the files where component dependencies are located, and is missing behavior information.

2.2.3 Syntactic Composability and Verification

Components are syntactically composable when their implementation details, such as parameter passing, external data access, and timing assumptions, are compatible [32,

88]. This means that components can properly connect and communicate in a common language. In other words, a component-based framework must ensure that components communicate at a technical level, i.e., components are correctly interconnected and the digital output from one component reaches the digital input of the component(s) to which it is connected [30]. Syntactic composability is closed under composition, i.e., the composition of two syntactically valid components forms a syntactically valid composition.

While several state-of-the-art component-based simulation frameworks provide for the context in which syntactic composability is achieved, very few actually verify the syntactic composability of the composed model. In the current context where components may be implemented by different vendors, it is important for the framework to independently verify that the composition is syntactically correct. This translates into verifying that all components are correctly connected and that their data passing mechanisms and time assumptions are compatible. Several factors that influence the automated verification of syntactic composability include the component abstraction and the composition rules among others. The component abstraction should describe the component communication and how it connects with other components in the composition. This can be achieved at a general level, through a framework-wide specification, or at a specialized component level. While a specialized component level description allows each component to describe its communication with outside components (e.g. the number of communication channels, type of data, etc.), it is more difficult to specialize than a framework-wide paradigm which imposes a finite number of communication channels and a standard data description format on all components. Next, a specification of composition rules to facilitate automated verification of syntactic composability is needed.

From the frameworks under analysis, both DEVSJava and OSA do not verify syn-

tactic composability. However, in both cases, syntactic composability could be easily verified under the assumption that the components implement the specific framework formalisms, DEVS and Fractal respectively. Current work that employs BOM verifies syntactic composability by matching between the method parameters in the composition scenario. This is a different approach as it deals with another meaning of syntax, similar to method signature [75] in software engineering. Its main disadvantage is that it does not focus on the actual connections of the components, since only the user specified scenario is employed in the verification without considering the discovered components. Petty and Weisel assume in their work that syntactic composability is achieved and verified beforehand [87]. More importantly, syntactic composability and verification is not possible using the Petty and Weisel formalism alone because the formalism does not contain any information about component communication and connection.

2.2.4 Semantic Composability and Validation

In *Semantic composability*, the data exchange between components must be meaningful, components must have the same understanding of the simulation context and of the underlying model assumptions, and the composed model must be valid in terms of the desired user objectives [12, 87, 88]. As such, models that are semantically valid are desired and should be produced by component-based frameworks [61, 88]. In simulation, validation is defined as "*the overall process of comparing the model and its behavior to the real system and its behavior*" [11]. Comparison ranges from *subjective* tests by system experts to *objective* tests by comparing the simulator data with data obtained from the real system. There is a well known distinction between the verification of a simulation model, in which we ensure that the simulator implements the simulation model correctly, and validation, in which we ensure that the correct model is developed. Extrapolating on verification and validation for composable simulations, we could say

that *verification* refers to checking *syntactic* composability because it ensures that the components (which are the basic building blocks of the composition) are correctly connected, whereas *validation* refers to ensuring that *semantic* composability is achieved.

The validation of semantic composability is not a trivial problem [11, 30, 88, 126]. Firstly, semantic composability is not a closed operation, meaning that semantically valid components do not necessarily form semantically valid compositions [10, 12, 30, 87]. Secondly, the interaction of reused components may result in emergent properties [49]. Thirdly, several perspectives must be considered when validating the interoperation of simulation components. The composed model can be validated from logical and temporal perspectives by considering properties such as safety and liveness of the composition behavior over time. In this context, dynamic component and composition behavior are difficult to formalize. More importantly, validity is generally not a fixed-point answer. There is a need to provide a formal measure of the degree of validation, a so-called "figure of merit" [61] to increase model credibility. From an implementation perspective, a key issue that influences the performance of semantic composability validation is the size of the composed model, in terms of the number of components and the number of attributes and states in the component representation, which can cause state space explosion [30, 121].

Current work on the formal validation of DEVS models represents DEVS models in the Z specification language [127]. A theorem proving tool based on Z such as Z/EVES [100] is used to verify the model and determine hidden properties. Ambiguities, conflicts and inconsistencies can be discovered in the specification. However, the Z specification language lacks time modeling, a most important attribute in DEVS models. As such, the validation process is incomplete. Validation is not performed in OSA, since the main focus of this research is on the composition execution. In current work employing BOM [75], a valid composition of discovered components is one in which

the sequence of actions or events is the same as or includes the sequence specified in the scenario. However, the somewhat informal validation process includes the composition and execution of discovered components in *all* possible combinations in order to be compared with the specified scenario. Moreover, a detailed execution scenario might not be available from the model composer.

As discussed above, the formal theory of composability validation proposed by Petty and Weisel allows for a composed simulation model to be checked for semantic validity. They propose a formalism in which a component is modeled as a mathematical function. The simulation of a composition is represented as a Labeled Transition System (LTS) [114] where nodes are model states, edges are function executions, and labels are model inputs. A composition is valid if and only if its simulation is close by a relation to the simulation of a perfect model. However, time is not modeled, which permits only a static representation of the composition and is not perfectly suited to simulation components with dynamic behavior. Furthermore, the LTS representation cannot consider complex composition with fork/join connectors.

Figure 2.6 summarizes our analysis of state-of-the-art component-based simulation frameworks and approaches, namely, the Discrete Event System Specification (DEVS), the Open Simulation Architecture (OSA), the Base Object Model (BOM), and Petty and Weisel's formal theory of composability. While most of the above approaches provide piecewise solutions to important issues of simulation composability, such as the validation of syntactic and semantic composability, they fail to provide an integrated framework for component-based simulation development. As discussed above, such an integrated framework is not simply a collection of individual solutions to important steps in the component-based simulation life-cycle, but a well-oiled ensemble in which all parts are designed to work together to achieve a maximum efficient process. This implies that all individual parts such as component representation, a process for

component discovery and selection, and semantic composability validation, have to be designed and implemented with the other steps in the component-based simulation life-cycle in mind. For example, an appropriate component representation has to be chosen with a correct level of abstraction to facilitate syntactic composability verification in which only a high level of abstraction is needed, and semantic composability validation, which requires a lower level of abstraction. As such, the studied approaches are not easily extended towards an integrated framework for component-based simulation model development. While DEVSJava and OSA are integrated component-based simu-

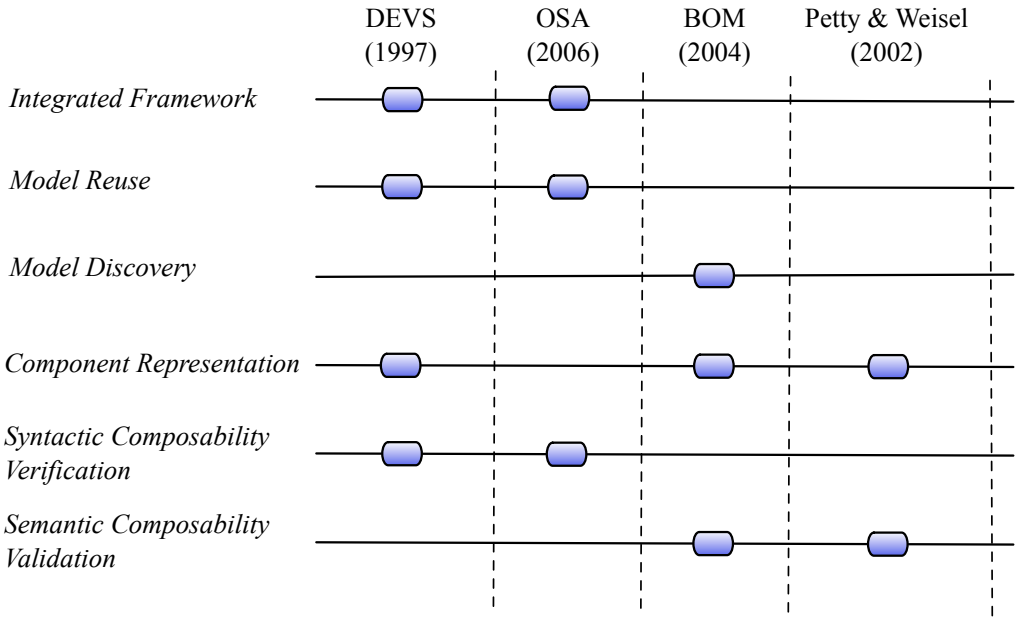


Figure 2.6: Comparison of Component-based Solutions

lation frameworks, model discovery, the verification of syntactic composability, and the validation of semantic composability are not addressed. On the other hand, approaches that employ BOM focus on model discovery and the validation of semantic composability based on a low-level simulation scenario. Lastly, Petty and Weisel's formal theory assumes that an integrated framework in which models are discovered and semantically composed exists beforehand. As such, the focus of their theory is only on the formal validation of semantic composability.

2.3 Summary

In this chapter we have studied four state-of-the-art approaches to component-based simulation development, namely, the Discrete Event System Specification (DEVS), the Open Simulation Architecture (OSA), the Base Object Model (BOM), and Petty and Weisel's formal theory of composability. We have analyzed the above approaches from different perspectives. Firstly, we analyzed if the above approaches offer an integrated framework for component-based simulation development, which takes a composed model from its conceptual stage, through model discovery and selection, the verification of syntactic composability, and the validation of semantic composability. The benefits of an integrated approach include increasing efficiency in the composed model development process, and a transparent user experience that facilitates adoption to a wider user community. Most of the studied approaches attempt to obtain an integrated framework but several design decisions prevent them from reaching this goal.

An important desired characteristic of component-based simulation model development is to achieve model reuse towards largely scalable models. This is achieved by most studied approaches, at different levels, such as the reuse of individual components or the reuse of components in larger, hierarchical compositions. However, towards model reuse, meaningful discovery and selection of reusable components is of paramount importance. This is not achieved at a satisfactory level by any of the studied approaches. One of the reasons for this drawback is the component representation, which was not designed with discovery in mind. Lastly, the validation of semantic composability is of paramount importance to increase model credibility. Of the studied approaches, Petty and Weisel's theory is the only one that focuses on semantic composability validation. However, the proposed Petty and Weisel formalism is theoretically elegant but cannot be adapted to real simulation components.

Chapter 3

Proposed Approach

Component-based modeling and simulation is a multi-step process in which the composed model follows a complex life-cycle, from a conceptual stage to a validated simulator that is ready for experimentation. On one hand, design decisions such as the level of component abstraction, the structure of the component repository, the approach to model discovery and selection, and the strategy employed in semantic composability validation, have a significant impact on the efficiency of the development life-cycle. On the other hand, a component-based simulation approach must express desired characteristics such as facilitating the development of large-scale simulations, component reuse, as well as the sharing of components across application domains. As such, efficient individual life-cycle step solutions must consider the impact of their associated design decisions on the overall attributes and context. More importantly, a piecewise approach, in which separate solutions for each step are integrated without considering the overall impact will lead to inconsistencies, difficulty in maintenance, and decreased efficiency. What is needed is an integrated approach for component-based modeling and simulation where various design decisions and solutions to crosscutting issues are integrated seamlessly towards an efficient process with reduced development costs and desired characteristics [61].

This chapter presents the design overview of CoDES (Composable Discrete-Event scalable Simulation) [118, 119, 122, 125], our proposed integrated approach for component-based simulation model development. Two main objectives drive the design of our proposed integrated approach. Firstly, we propose to support the entire component-based simulation life-cycle, from conceptual model design to model verification and validation. Secondly, to increase model credibility, we propose a practical strategy for semantic validation that includes a rigorous formal approach with higher validation accuracy. To facilitate component sharing and reuse, the proposed approach is designed for scalable deployment on Internet-based infrastructures such as web services, grids, and clouds.

We structured our approach and this thesis along the following component-based modeling and simulation life-cycle, namely, conceptual model definition, model discovery and selection, syntactic composability verification, and semantic composability validation. This chapter is organized in two parts. Firstly, in the design of the CoDES framework, we address several issues that crosscut the component-based simulation modeling life-cycle, such as component abstraction, and domain knowledge representation among others. Secondly, we present key design problems and their solutions, such as the speed of model discovery and selection, the problem of partial matches, and how to increase the shared model credibility through a formal semantic validation process.

3.1 Crosscutting Design Issues

As discussed above, the life-cycle of a component-based simulation starts with conceptual model definition, followed by model discovery and selection, syntactic composability verification, and semantic composability validation. Figure 3.1 presents several issues that crosscut these life-cycle steps. These issues are defined as “crosscutting”

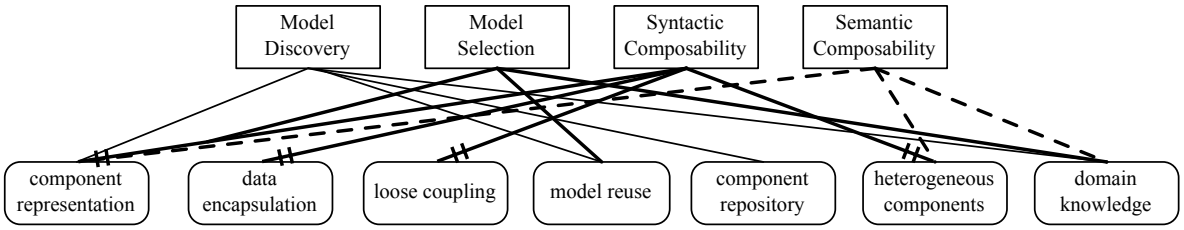


Figure 3.1: Crosscutting Issues in Component-based Simulation Frameworks

because their solution impacts different life-cycle steps in various ways. For example, a black-box *component representation* in which the structure and other information about the component is not accessible to the framework, does not influence the process of *syntactic composability* verification, but may decrease the accuracy of *model discovery and selection*. The crosscutting issues that we identify include component representation, data encapsulation, loose coupling, model reuse, the existence of heterogeneous components, the representation of domain knowledge to facilitate model discovery and semantic composability validation, as well as the existence of a component repository organized to support heterogeneous components with respect to application domain and geographical location. Our approach addresses these issues to provide a solid foundation for seamless component-based simulation model development. Our solutions are presented and analyzed below.

3.1.1 Component Representation

The building block of a component-based approach is the simulation component. In CoDES, we define a simulation component as follows.

Definition 1 (Simulation Component). A simulation component is a stateful, reusable, self-contained unit that meets diverse user requirements in different simulation contexts. It interacts with other components through well-defined connectors and based on the received input, its behavior changes over time.

To facilitate composition and reasoning, simulation components must have a standard internal representation in the framework, to which all components must adhere [61, 88]. The choice of *component representation* has major implications on the functionality of the CoDES modules in Figure 3.5 and as such cannot be treated lightly. The component representation must be independent from the programming language in which the component is developed. Furthermore, it must offer enough details about the component attributes and functionality without being too complex. A component representation that is too complex may be unreadable by human users and might hinder the performance of computer algorithms that employ it [19]. On the other hand, a very simple component representation may be too abstract for computers to process meaningfully and might not meet the specific level of detail required by component users.

The abstraction of simulation components dictates the reusability of model components as part of a larger simulation model, as well as the cost of validation. In a black-box abstraction, component details, such as the hierarchical structure, and attributes and behavior, are not visible, and a simulation component is described only in terms of input and output data (usually in forms of messages) [123]. In contrast, a white-box abstraction exposes the component structure and its internal details. While the black-box abstraction is computationally less expensive than a white-box approach for both syntactic and semantic validation, our preliminary studies reveal that this may result in a loss of validation accuracy and model credibility [121]. However, white-box representation increases validation costs and limits the size and scalability of composable models.

Besides the black-box/white-box trade-off discussed above, another trade-off arises between the various levels of modeling resolution captured by the component representation [79]. For example, a detailed and accurate representation of the exact component behavior in all possible usage contexts is desired for meaningful discovery and accurate

validation. However, such a representation is very difficult to define and might lead to overspecification [19].

We propose a *meta-component* description in terms of the *attributes* and *behavior* of a component. The component attributes are classified into *mandatory attributes* and *specific attributes*. Mandatory attributes are common to all components and include information such as the component name, author, location, usage history, etc. Specific attributes include information related to individual components, e. g. "serviceTime" and "interArrivalTime" for a Server component. The component behavior is represented in two ways, *externally*, with respect to the data that the component sends and receives from other components in the composition, and *internally*, as a timed state machine. The component behavior describes the data that it receives and outputs as a set of states. The transitions between states are defined as a set of triggers expressed in terms of input, time and conditions. As such, our proposed implementation strikes a balance between different levels of modeling. For example, neighboring components are interested in what data the component sends and receives. On the other hand, model discovery and selection requires component attributes to perform matching, whereas the validation of semantic composability requires detailed information about the component behavior.

More formally, a meta-component C_i is defined as follows:

Definition 2 (Meta-component). A meta-component represents a simulation component C_i as a tuple:

$$C_i = \langle R, A_i, B_i \rangle \tag{3.1}$$

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

$$[I_l]S_p[\Delta t] \xrightarrow{Cond_n} S_t[O_l][A_m] \tag{3.2}$$

where I_l is the set of input data; S_p is the current state; Δt is the transition duration; $Cond_n$ 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 modified attributes after the state change.

The data that a component sends and receives has to be properly specified in order to ensure that the framework can establish data compatibility between components in the composition. We propose to describe the data sent and received by a component using *data constraints* which are descriptions of data that are semantically enriched with descriptions and relations defined as in our proposed COSMO (COmponent Simulation and Modeling Ontology) ontology. These constraints include the *type* of data, *range* of its values, *origin*, *destination*, and 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. To describe non-primitive data types, we propose a general *class* data constraint. Our proposed data constraints are used in discovery and selection to identify the components most suitable to a user query, and in semantic composability validation to flag invalid data passing through the composition according to user-specified validity points.

As discussed above, the meta-component is an accurate representation of the component implementation and is used throughout the CoDES framework. The meta-component is a description provided by the component developer and is attached to the implementation when the component is added to the CoDES repository. The terms "component" and "meta-component" will be used interchangeably in the remainder of this thesis.

It is important for all simulation components to be represented in a machine readable format to which all components in the framework adhere. Towards this issue, we

propose COML, a markup language for representing CoDES components. COML is described in XML Schema. When new components are added to the CoDES framework, their XML meta-component definition is checked against the COML schema. COML includes tags such as: $mandAtt$ and $specAtt$ to describe mandatory and component specific attributes, $behavior\ transitions$, $input$, $output$ to describe the component behavior, including states, transitions, constraints on the input, and output, $topology$ to describe the topology of model components and standalone simulators, $prodRule$ to describe the standalone simulator's production rule, etc. Figure 3.2 presents excerpts from the COML Schema for base components.

To facilitate model discovery and selection, the entities in the COML file are described in COSMO, our proposed component-based ontology. As such, the component representation together with the COSMO ontology provide extensive descriptions without overspecification. COML also provides a schema for representing the input/output data exchanged between components. This guarantees that communication between components is performed in a correct and standardized format.

3.1.2 Data Encapsulation and Loose Coupling

Data encapsulation or separation of concerns [36, 93] is the process of breaking a computer program into parts that overlap as little as possible or not at all. In a component-based context, design decisions and implementation issues that are particular to a component do not affect the implementation of other components in the composition. Complex compositions are thus easier to understand, design and maintain. In the context of component-based simulation in the CoDES framework, we consider a simulation component to be a *black-box* with an input and/or an output communication channel. As such, the component's implementation is hidden from other components in the composition. The user can change the component execution by modifying component at-

```
<element name="component"> <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 & final state name for each transition shld 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 3.2: Component Representation in COML

tributes, which in turn change the component behavior.

To facilitate *loose coupling* [111], the CoDES framework proposes a component-connector paradigm to which all CoDES components must abide [118]. In the component-connector paradigm, components are interconnected by well-defined connectors. A connector performs message and data passing among components. As such components need only implement communication protocols with standard connectors and not with other components. Messages leaving components are timestamped and the connector guarantees FIFO delivery of messages to the destination components. As shown in Figure 3.3, connectors are divided into *one-to-one* (Connect) 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 is summarized below.

Definition 3 (Black-box Component-Connector Paradigm). CoDES components view each other as black-boxes with one input and/or output communication channel. Components are interconnected using framework connectors, which are classified in one-to-one, fork, and join. Connectors are responsible for data passing between components in the form of messages.

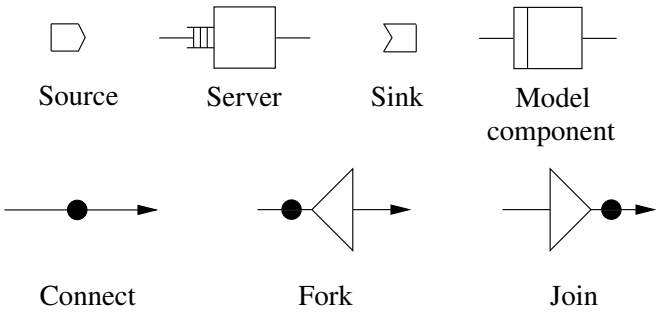


Figure 3.3: CoDES Components and Connectors

3.1.3 Component Organization

To achieve knowledge sharing to a wider user community, a component-based simulation approach must consider *heterogeneous* components. Component heterogeneity refers mainly to components developed for specific application domains, but may also refer to the component location. Examples of application domain based simulation frameworks include electronic systems [33], thermofluid systems [102], mechatronics [25], and network [106] simulations. While component-based approaches for specific application domains are perceived to be easier to develop and achieve greater depth into the coverage of a particular application domain, approaches that generalize and share components across application domains will increase the level of component reusability, and facilitate the development of larger and more complex simulations at lower costs. A main challenge is to achieve component sharing both in *breadth* (many domains) and in *depth* (detailed specific domains), and at the same time allow for facile cross-domain component integration, and the validation of the heterogeneous composed artifact [61].

The second type of heterogeneity refers to the *location* of the component implementations. Component-based simulations can be developed using components from distributed repositories that can reside in different administrative domains. Moreover, there is an increasing trend in using the Internet as an infrastructure for the discovery and (re)use of shared resources, which promises to advance knowledge sharing to a wider simulation community.

Component sharing *within* and *across* application domains leads to expensive model discovery and semantic composability validation in the absence of a hierarchical component organization. This is because the lack of hierarchical component organization incurs a large number of futile comparisons in model discovery and selection. In semantic composability validation, a hierarchical component organization can reduce the number of ontology queries and thus decrease the validation runtime [121]. We propose

a hierarchical component organization as shown in Figure 3.4. The CoDES component

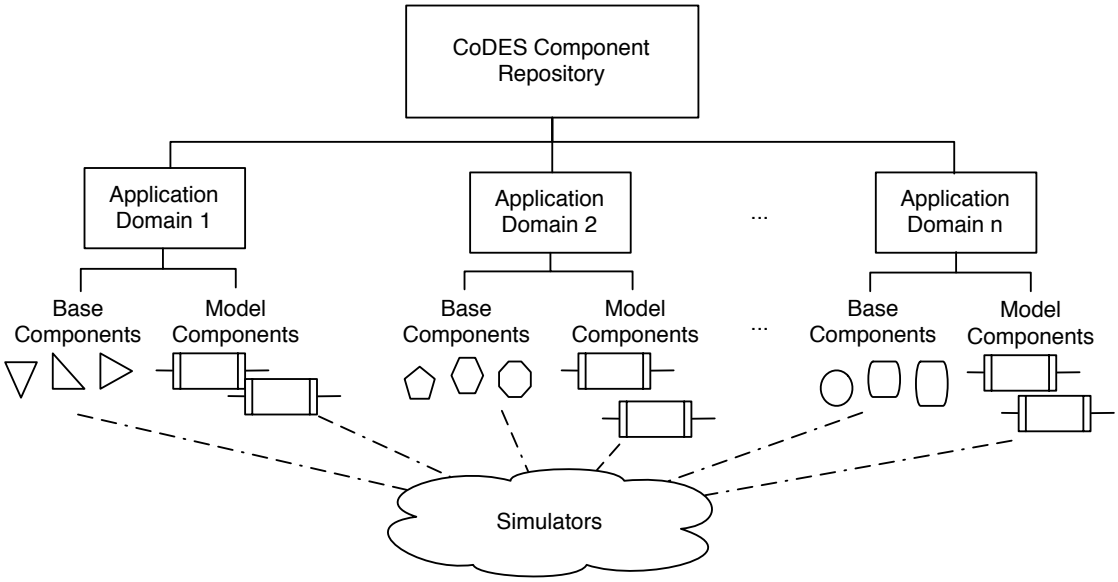


Figure 3.4: Hierarchical Component Organization

repository is organized in three logical categories, *base components* specific to each application domain, *model components* developed from base components and other model components, and *standalone simulators* that can contain both model and base components. Our component definitions are presented below.

Definition 4 (Base Component). A base component is a well-defined atomic block that represents a fundamental entity in an application domain.

Definition 5 (Model Component). A model component is composed from base components and other shared model components.

Base components form the basic building block in a conceptual component-based model and each component is viewed as a black-box with in and/or out communication channels. Model components, each represented as a black-box with both in and out communication channels, can be reused across application domains. We discuss in Section 3.1.4 how model components are created.

For example, the Queueing Networks application domain has three base components, namely, source, server, and sink as shown in Figure 3.3. *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 major advantage of the separation into base components specific to each application domain is that intrinsic knowledge about the application domain is captured without the necessity of any formalism. However, this somewhat informal knowledge is not sufficient, in particular in model discovery and selection and in semantic composability validation. For example, in model discovery and selection, it is rarely the case that there exist exact matches to the user query, and as such partial matches need to be computed. This is done by estimating the similarity between the user query attributes and behavior and the attributes and behavior of the repository components. On the other hand, in semantic composability validation, the degree of compatibility between the input and output data of communicating components needs to be computed. Furthermore, when comparing the execution of the composed model with that of the ideal model desired by the model composer, several degree of closeness appear and need to be measured. As such, there is a need to express *domain and component knowledge* in an unambiguous, standardized format that is accessible to humans and computer programs.

An *ontology* is an organized knowledge representation to capture object information in a particular domain [112], in formats readable by humans and computers alike. On-

tologies are conceptual models that capture and explain the vocabulary used in semantic applications guaranteeing communication free of ambiguities [17]. When applied to the modeling and simulation domain, ontologies facilitate model discovery and integration and the development of formal methods for simulation and modeling [71, 110]. Ontologies can be used to express syntax and semantics to facilitate communication and allow for automated semantic checking. Furthermore, they are employed to express the resource discovery request and determine whether the discovered model is reusable. We propose the COSMO ontology to meaningfully represent component-based simulation knowledge, as well as application domain specific information. We discuss COSMO in Chapter 5.

3.1.4 Model Reuse

The reuse of simulation models is of paramount importance to decrease the cost of development of large simulation models [61, 12]. Simulation model reuse requires methodologies for abstraction, to facilitate storing and retrieving of simulation models from *component repositories*, for selection of simulators or model components, and for the integration of reused model components in new simulators. Towards model reuse, component repositories have a powerful impact on the acceptance of a component-based approach [31, 61, 80]. The structure of a component repository must consider both distributed and local settings, and must facilitate a fast, scalable, discovery service. Another important factor in achieving simulation model reuse is the component abstraction, which facilitates discovery and reasoning about the composed model from reused components. The abstraction of a component as a meta-component facilitates the process of discovery, selection, and integration, because it offers an accurate description of the component implementation.

In the CoDES framework, we propose three levels of model reuse. Firstly, we pro-

pose and facilitate the reuse of a developed simulation model "as-is". This implies that a composed simulator is saved into the repository after it has been developed for it to be re-executed and re-used at a later time. This is in accordance with current work in composability which focuses mostly on achieving reuse on an "as-is" basis [29, 69, 77]. While the simulator is saved into a repository for future reuse, there exists no possibility of inclusion in larger models, or even modification to suit user needs by changing attribute values. While this approach still reaps some of the advantages of model reuse, it does not facilitate the development of larger models and limits the growth of the repository to a wider user community. To address this drawback, we secondly propose the *reuse of base components* in different compositions. A reused base component can be part of many compositions. It is stored in a component repository and can be retrieved whenever is necessary. This type of reuse captures the true philosophy of reuse and reaps the full advantage of composability, facilitating the development of larger simulation models. On an even wider scale, we propose *model components* developed from base and other model components as the third type of model reuse. The three types of reuse, namely, standalone simulators, base components, and model components, facilitate the development of large simulation models at various scales to increase knowledge sharing, model scalability, and community reach.

As discussed above, the way a component is represented in a meta-component is highly important for the process of model reuse. As described in Section 3.1.1, the base components are abstracted as black-boxes, with their behavior described in a meta-component using state machines and attributes. The abstraction of model components dictates the reusability of model components as part of a larger simulation model and the cost of validation. In a black-box abstraction, component details, including their hierarchical structure, attributes and behavior, are hidden, and a model component is described only in terms of input and output. In contrast, a white-box abstraction ex-

poses the model component structure and its internal details. While the black-box abstraction is computationally less expensive for both syntactic and semantic validation, our preliminary studies reveal that this may result in a loss of validation accuracy and model credibility [121]. However, white-box representation increases validation costs and limits the size and scalability of composable models.

Currently, a model component is viewed as a black-box with in and out communication channels. Information about the input and output that can be received/sent through these communication channels is also stored. In a future work, we intend to study the trade-off between a black-box and a white-box approach to enhance the reusability of the information stored in the model components towards faster model discovery and selection and better reasoning in semantic composability validation.

To facilitate the three types of reuse described above, we distinguish between two types of information, syntactic, which describes the component in terms of attributes and structure, and semantic, which describes the component behavior. Both syntactic and semantic information are employed in model discovery. Semantic information is also employed in the process of semantic composability validation. Table 3.1 presents the information content of the meta-component abstractions. For base and model components, the syntactic information saved in the meta-component refers to the component's mandatory attributes (e.g. name, author), and specific attributes (e.g. "interArrivalTime"). The syntactic information pertaining to a standalone simulator contains the production string obtained form the composition grammar, and the simulator's topol-

<table>
<tr><th>Abstraction</th><th>Base Component</th><th>Standalone Simulator</th><th>Model Component</th></tr>
<tr><td>Syntactic</td><td>attributes
(name, description, iaTime etc.)</td><td>production string
topology</td><td>attributes
(name, description, author etc.)</td></tr>
<tr><td rowspan="3">Semantic</td><td>behavior</td><td rowspan="2">all meta-components</td><td>-</td></tr>
<tr><td rowspan="2">external : I/O constraints
internal : state machine</td><td>behavior</td></tr>
<tr><td>validation information</td><td>external : I/O constraints
internal : -</td></tr>
</table>

Table 3.1: Abstractions Towards the Reuse of Model Components

ogy. The production string is a linear arrangement of the components' types according to their position on the graphical screen, and is accepted by the composition grammar specific to the simulator's application domain. The topology describes in detail the connections between the components in the simulator. Since there can be more than one implementation for the same production string, the production string and the topology uniquely identify the simulators in the repository. The semantic information encapsulated in the meta-component of a base component refers to the component's behavior, both *external*, as data constraints on the input and/or output data, and *internal*, as the base component's state machine. For a standalone simulator, the semantic information contains all the meta-component definitions of all composing components. Furthermore, the validity information resulted from the semantic composability validation process is also saved. The only semantic information saved for a model component is given by the input/output data constraints for the components that were at the cut-off points when the model component was created from a validated simulator as discussed above. This is in accordance with a black-box view of the component as proposed above.

In the CoDES framework, a model component that is saved into the repository must have both "in" and "out" communication channels to facilitate its reuse in the maximum number of compositions. This is obtained by stripping off some components from a semantically valid standalone simulator, according to some framework or user defined cut-off points, and wrapping the remaining components using a framework *wrapper*. As such, information about all composing components is retained, but only partial validation information is saved since some of the information might depend on the components that were stripped off. Furthermore, even though information about the constraints on the input and the output is saved as external behavior, no information about the model component's state machine can be obtained.

3.2 Design Overview

Figure 3.5 presents a block diagram of the CoDES framework, highlighting the interactions between the user and the framework throughout the life-cycle of the composed simulation model.

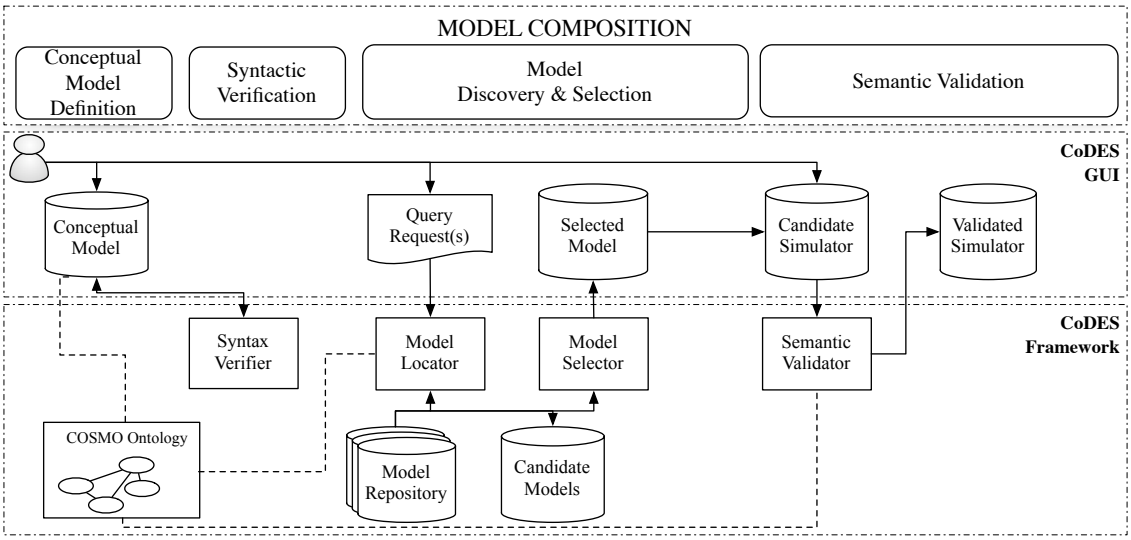


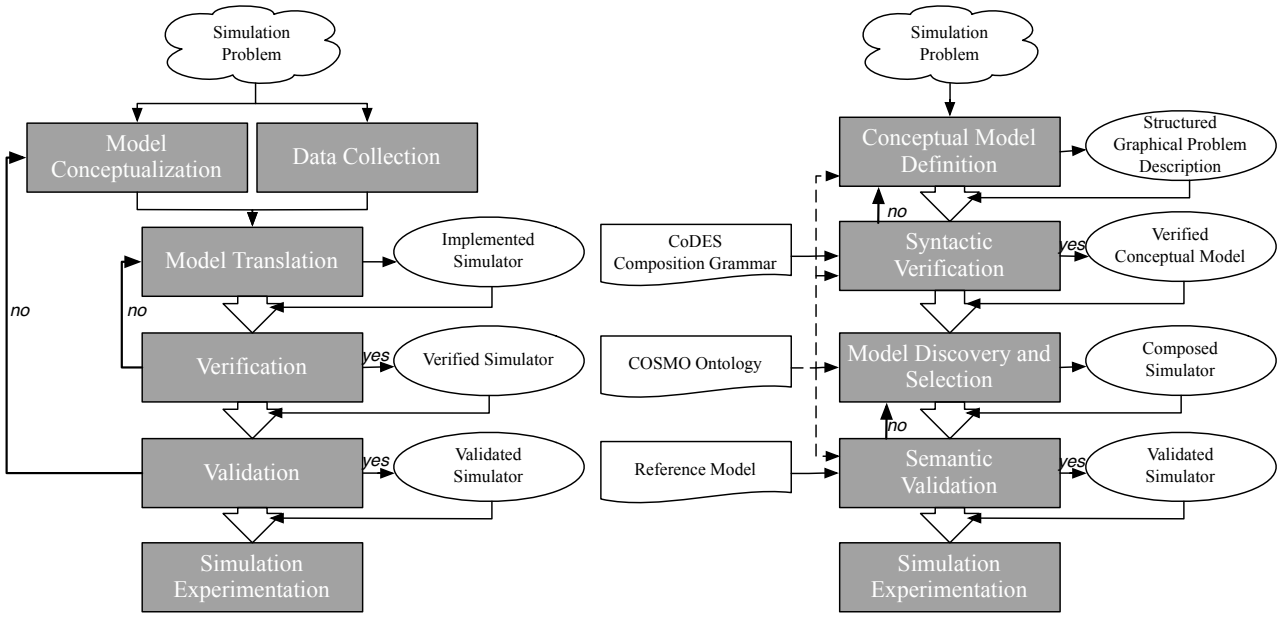
Figure 3.5: High-level Overview of the Design of CoDES

We propose a new life-cycle for a component-based simulation study in the CoDES framework, with the following steps in this order, namely *Conceptual Model Definition*, *Syntactic Verification*, *Model Discovery and Selection*, and *Semantic Validation*. A new simulation model is created using a graphical environment by drag-and-dropping icons of conceptual components on a drawing panel and linking the components using connectors. This is a *conceptual model* because the icons describe the simulation problem conceptually, and represent entities *without* an attached implementation. Syntactic composability is next verified in the *Syntactic Verification* step. In *Model Discovery and Selection*, the *Model Locator* module queries the model repository for the entire model or for individual components, resulting in one or more *candidate models*. The best candidate model is automatically selected. In *Semantic Validation*, the user sets the

desired attribute values for the components in the selected model, resulting in a *candidate simulator*. The candidate simulator is then semantically validated resulting in a *validated simulator* that can be subsequently executed.

3.2.1 Integrated Component-based Approach

We propose an integrated life-cycle for the development of a component-based model in the CoDES framework, consisting of four key steps as shown in Figure 3.6(b), namely *Conceptual Model Definition*, *Syntactic Verification*, *Model Discovery and Selection*, and *Semantic Validation*. Initially, a simulation problem is translated into a graphi-



(a) Traditional Simulation Life-cycle (adapted from Banks et al., 2005)

(b) Proposed Component-based Simulation Life-cycle

Figure 3.6: Life-cycle of a Simulation Study

cal conceptual model using component icons. The *Conceptual Model Definition* phase employs component icons without an attached implementation. The component icons are dragged and dropped on a drawing panel and subsequently connected using well-defined framework connectors, such as "join" and "fork". Next, the syntactic composability of the conceptual model is verified using our proposed EBNF composition gram-

mar in the *Syntactic Verification* stage. If the conceptual model is syntactically correct, the component implementation can be discovered in the *Model Discovery and Selection* step. The discovery service ranks the repository components based on the simulation developer component query. The repository components are ranked based on semantic domain knowledge expressed in our proposed COSMO ontology. Lastly, the semantic composability of the composed model is validated in the *Semantic Validation* step.

As shown in Figure 3.6, our proposed life-cycle differs slightly from current simulation approaches, where component implementations are first discovered, connected, and their connection is subsequently verified [61, 133], we verify the syntactic composability of the conceptual model *before* model discovery. This ensures that the model is correct before the costly discovery process is performed. Chapter 8 presents an evaluation of the cost of model discovery and selection. Syntactic composability verification performed before model discovery is equivalent to the verification performed after discovery and selection in traditional approaches. This is because our proposed approach guarantees that: (i) the number of communication channels of the component implementation is equal to the number of channels of the icon representation, and (ii) components and their communication are represented consistently by the same schema. From (i) it follows that the component connection will be the same regardless of whether the components are only conceptual icons or have attached implementations. Next, (ii) guarantees that all repository components use the same standardized syntax for communication.

Another component-based simulation life-cycle proposes the specification of components in XML, from which component code can be generated [98]. The proposed process has four important stages, namely the description of a component in XML, code generation from the XML specification, composition, and simulation execution. This approach is platform independent and can be added to simulation systems such as

James II [52]. Moreover, the XML-based specification facilitates syntactic composition. However, the validation of the composed model is not discussed. The key assumption in this approach is that a complete specification of the component, probably provided by the model composer, exists to facilitate the component code generation. This bypasses one of the recognized advantages of component-based approaches, which is the reuse of previously developed components to reduce costs. In contrast, we adopt a reverse view of the composition process, assuming that component implementations exist in a component repository, and that their behavior and characteristics are captured in a meta-component described in XML. We propose methods for the discovery and selection of component implementations. Moreover, we propose solutions for the verification and validation of the composed model.

3.2.2 Formal Time-based Semantic Validation

Studies of semantic composability validation show that model validity is not a fixed point answer and there are many valid models but with different degrees of validity [76, 88, 119, 122]. Current approaches to validate composed models such as the formal theory of composability, are theoretically elegant but not scalable and thus are not practical to implement [88] or suffer from validation state-space explosion and thus do not scale [77, 127]. These trade-offs are inherent in the validation of semantic composability because of the many facets of validation and the ambiguity of the definition of validity. Moreover, semantic validation is a costly process. Firstly, many aspects can be considered when discussing semantic validation, such as *logical*, with respect to safety and liveness, *temporal*, with respect to the behavior of the composed model over time, and *formal*, with respect to the formal guarantees offered by the validation process to increase model credibility. Secondly, there exist many definitions of a valid composed simulation model, which depend on the simulation objective desired by the user. These

definitions are difficult to specify in a formal, generalized manner that allows for the implementation of an automated validation process. With this in mind, we propose a layered approach to validation in which we consider the aspects defined above and provide definitions of validity from a user and a formal perspective.

Based on our composability studies of reusing base components, we observe that there are more invalid than valid models. As such, our strategy is to check for invalid models. This is less costly on average if we employ a dual-step *deny validity* approach. Figure 3.7 presents a high-level overview of our proposed validation process.

In the CoDES framework, only models that are syntactically correct reach the model discovery and selection step. This is facilitated by our proposed life-cycle, which relies on our standard component representation and our proposed component-connector paradigm. Next, the syntactically correct conceptual model is discovered in model discovery and selection, and the composed model is semantically validated.

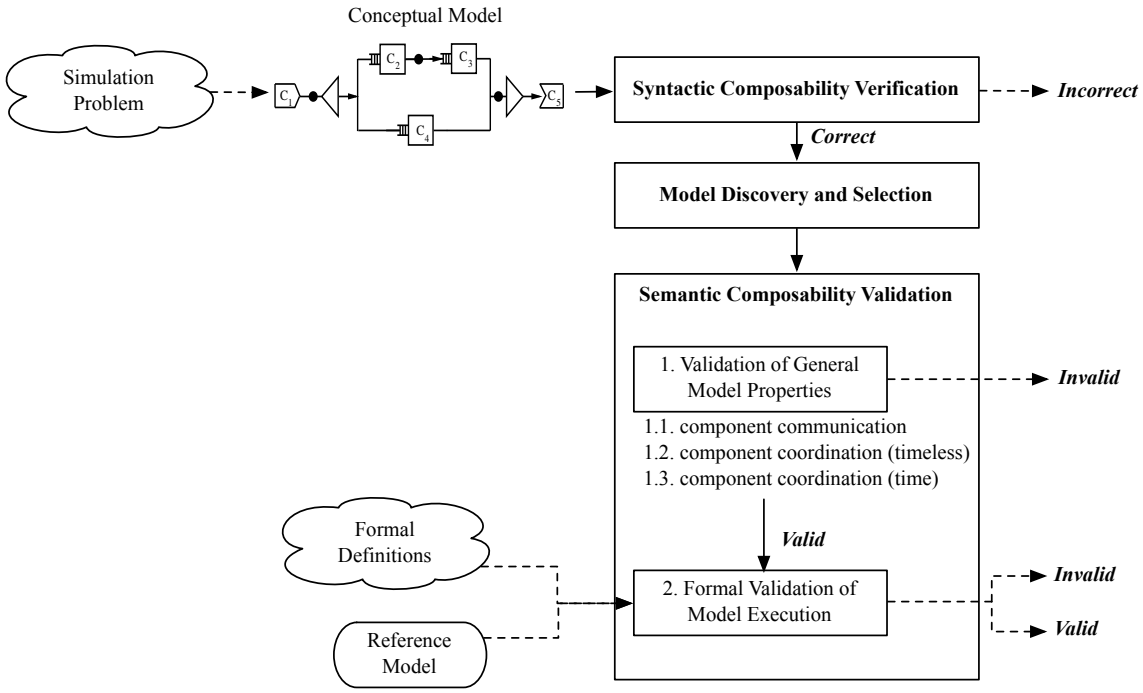


Figure 3.7: High Level Overview of Semantic Composability Validation

In the validation of semantic composability, invalid models are eliminated by checking general model properties for both instantaneous and timed transitions. However, a composed model with valid model properties may not be the exact model desired by the user. While the composed model might have desired semantic properties, it might not be in accordance with the real or the ideal system desired by the user. As such, we perform model execution validation by comparing the execution of the composed model with the execution of a reference model, which stands for an ideal representation of the real system.

Towards this comparison, we introduce a novel time-based formalism where a simulation component is represented as a function of its states over time. Based on our formal definitions of composition, simulation, and validity, we propose a formal validation process in which the main consideration is a simulation component that evolves over time by changing its state throughout the composition execution. This is in contrast with software engineering approaches where the components are often stateless artifacts in which time and states are not important. The validation process is presented in detail in Chapter 6 and Chapter 7.

3.3 Summary

In the design of CoDES, our proposed framework for component-based modeling and simulation, we focus on two key issues.

Firstly, the reuse of simulation components promises reduced development time and increased scalability. We propose a component-based approach that facilitates model reuse through a four-step life-cycle in which all steps are integrated seamlessly with increased efficiency and runtime cost gains. Our proposed life-cycle is supported by solutions to several crosscutting issues, including among many: component representation, data encapsulation, loose coupling, heterogeneous components, representing do-

main knowledge, and a component repository that facilitates reuse and composition. To achieve component reuse across many application domains, our novel approach divides a simulation problem space into different compartmented application domains, which are subsequently described using composition grammars and ontologies. This new perspective reduces the simulation problem search space to achieve greater scalability and efficiency in syntactic composability verification, ontology reasoning in model discovery and selection, and semantic composability validation. Furthermore, in contrast to current approaches, we propose a life-cycle in which syntactic composability is performed *before* the costly model discovery and selection to increase efficiency.

Secondly, our approach focuses on the semantic validation of the composed artifact to increase model credibility. Since validity is a costly process that does not have fixed point answer, we propose a novel *deny validity* approach with incremental accuracy and cost. Firstly, invalid models are eliminated by checking general model properties for both instantaneous and timed transitions. This is less costly to identify and discard invalid models. We propose generalized and user-defined safety and liveness properties to cover various validation perspectives. If the composed model is not discarded in this test, model execution validation compares the execution of the composed model with that of the real system. In this step, we introduce a novel time-based formalism where a simulation component is represented as a function of its states over time. Using this formalism, we provide definitions of composition, simulation, and validity. These form the foundation of our proposed formal validation process that considers dynamic and stateful behavior, as well as semantically-sugared descriptions, all of which are of paramount importance for component-based simulations.

Chapter 4

Model Composition and Verification

In the first step of the composition life-cycle, the simulation developer (hereafter referred to as model composer) specifies the composed model by drawing it using component icons. The model is then compiled and executed. For the model to execute properly, components must interoperate and the data exchanged must be compatible. This is called syntactic composability [30, 88]. In our proposed life-cycle, this sequence of events translates to a conceptual model definition step followed by syntactic composability verification.

In current approaches to component-based simulation modeling, the simulation developer first designs a *conceptual simulation model* that mimics the real system under study. For example in the Arena simulation software [62], icons that model different real-world entities are placed on a drawing panel and connected to form a graphical representation of the conceptual model. The entity parameters are defined and the simulation model is subsequently executed. The conceptual model is created in the problem modeling phase of the simulation study. It describes simulation objectives, and consists of assumptions, objectives, algorithms, relationships and data (without implementation) that describe how the simulation developer understands what is being represented by the simulation [84, 96]. In most component-based simulation modeling approaches

[29, 50], the conceptual model if it explicitly exists, as is the case of OSA [29], translates into a collection of user-developed components, as well as reused components that have a hierarchical structure. To reuse components, the simulation developer must perform model discovery and selection. This process can be compute-intensive when the model repository is large, and the discovered components may not be an exact match to the user requests. Next, if the discovered components turn out to be incompatible or the composition structure is incorrect, the entire sequence "conceptual model definition - model discovery and selection - syntactic composability verification" has to be repeated and becomes iterative and costly.

In this chapter, we propose the verification of syntactic composability *before* model discovery and selection to decrease the computation costs. We propose a formalism based on compositional grammars to guide the definition of the conceptual model and facilitate syntactic verification. We first present an overview of current approaches to model composition and verification. Next, we present our proposed approach to model composition and verification, and exemplify using a composed simulator of a single-server queue. We perform a theoretical and experimental analysis of syntactic composability verification to showcase the benefits and scalability of our proposed approach.

4.1 Current Approaches

Figure 4.1 presents current approaches in model composition and verification in component-based simulations. Informally, syntactic composability verification is the process of checking that the components in the composed model are connected correctly and can interoperate. We outline several points of interest towards model composition and verification, namely *data encapsulation*, *component connection*, *composition representation*, and automated *syntactic composability verification*. Data encapsulation is important for syntactic composability because it ensures that implementation details pertain-

ing to each component are hidden from other components in the composition. This separation ensures that simulation components can be reused in a variety of contexts with minimum modifications. Component connection analyzes how components are connected in the composition. Here, a simple approach that guarantees scalability is desired, but usually cannot be obtained because of framework-wide factors such as the component model implemented by the framework. The composition representation can facilitate or hinder the automated verification of syntactic composability, depending on the level of abstraction. Lastly, the syntactic composability verification analyzes how verification is performed. In particular, automated syntactic composability verification is important when components are developed in various locations by different vendors and a fast, scalable, and repeatable framework-specific guarantee is needed. In the fol-

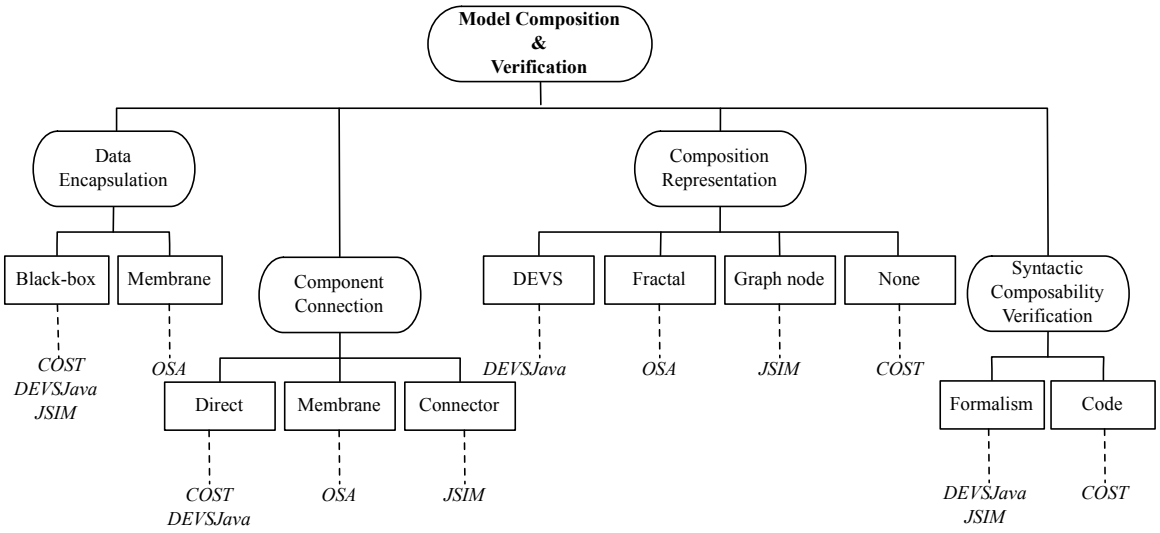


Figure 4.1: Approaches to Syntactic Composability

lowing, we focus on four state-of-the-art approaches, namely DEVSJava, COST, JSIM, and OSA.

Data Encapsulation

All current approaches to syntactic composability separate components in the compo-

sition. Components are mostly abstracted as black-box with input and output communication port*s*. In OSA, components offer a glass-box view of their inner structure by providing access to the sub-component methods through a *membrane*. A black-box view of components is advantageous because it hides inner component details (such as behavior and structure) from other components with whom the component interacts in the composition. This prevents overspecification and facilitates automated reasoning about syntactic composability with decreased costs. However, more details about the structure of the hierarchical components are needed for more accurate model discovery and selection and semantic composability validation. These are offered by a glass-box view of the component, but also incur higher costs in discovery and validation because of state space explosion caused by a large number of sub-components that are considered. This is a simple example of a design decision in one life-cycle step that can influence negatively the performance of other life-cycle steps.

Component Connection

Current approaches propose several methods of component connection, such as *direct*, *membrane*, and through *connectors*. In *direct* component interconnection, components are connected in the simulation code using methods published in their interfaces. A major disadvantage of this approach is that it incurs on the components the added overhead of inter-component communication. For example, for the COST framework, inter-component communication is done by employing specific C++ constructs such as templates and function pointers. This means that a component implementation must also be aware of other component implementations in order to properly communicate. Similarly, component communication in OSA is done through a component *membrane* that is part of the component. The membrane provides to the other components the exact methods that the component can execute, and handles time management within

the component as well as the communication with the rest of the components in the composition. Only in JSIM the component inter-connection is done through framework *connectors*. Connectors are advantageous because component implementations need focus only on the communication with standard framework connectors. In contrast, a non-connector-based approach would require the component implementation to cater for all possible implementation of the components with which it could communicate.

Composition Representation

The composition representation is of paramount importance towards both the verification of syntactic composability and the validation of semantic composability. Ideally, a formalism of some kind is desired to represent the composition. Such formalism, containing well-defined rules, permits reasoning about the composition, which in turn helps identify design errors and increases the credibility of the composed model. In our study, we have encountered several formalisms, namely the Discrete Event System Specification (DEVS) discussed in Chapter 2, the Fractal component model, and graph nodes. Components in DEVSJava implement the DEVS formalism, and the coupled DEVS formalism represents the entire composition. In JSIM, a component is represented as a graph node and the composition is a graph. An OSA component is represented as a Fractal component as discussed in Chapter 2. No formalism guides the composition of COST components, which is mostly an ad-hoc effort.

Syntactic Composability Verification

Components are syntactically composable if they are correctly connected and their input and output data is compatible. The verification of syntactic composability refers to checking that the component connection is correct and that the data exchanged between components is compatible. Several factors influence the runtime and credibility of the

syntactic composability verification process. Factors that influence the runtime include the number of components in the composition, the number and the types of connectors, and the number of attributes per component method.

The main factor that influences the credibility of the syntactic composability verification process is the formalism used to guide the verification. Syntactic verification can be performed either by directly compiling the entire composition source code, or by verifying the composition formalism to which the composition adheres. On one hand, the verification of a composition formalism offers greater credibility at the cost of accuracy loss through abstraction, since it is difficult to design a composition formalism that captures all the intrinsic details of a specific application domain. The level of abstraction is important, especially for a framework that caters for many application domains with entities with diverse granularities. On the other hand, compiling the composition source code can result in errors in connection (e.g. null pointer assignments) that are evident only at runtime. Lastly, the existence of a composition formalism guides the definition of the conceptual model and aides in its representation for discovery and validation. While to the best of our knowledge syntactic composability verification using the DEVS formalism is not implemented in DEVSJava, this can be easily done. Another framework in which syntactic composability is verified using a formalism is JSIM. In JSIM, a component is represented as a graph node and the composition is syntactically verified using graph algorithms. OSA implements the Fractal component model but does not verify syntactic composability. In the case of COST, syntactic composability verification is reduced to compiling the source code.

4.2 Proposed Syntactic Composability Approach

Syntactic composability is defined as interoperability between components in which *"the digital output from one component can be read as the digital input to the other"*

[30]. Syntactic composability requires that the components be developed in such a way that their implementation details, such as parameter passing mechanisms, external data accesses, and timing assumptions are compatible for all of the different configurations that might be composed. An important point to highlight is that syntactic composability does not mean a one-time, static effort of component interoperability. The components must interoperate regardless of the composition structure, allowing various configurations to satisfy different user requirements [30].

Figure 4.2 presents an overview of our proposed approach for syntactic composability and verification. The simulation model composer first conceptualizes the simulation model through a drawing in a Graphical User Interface (GUI), using icons of base and model components. The icons form the basic building blocks of the conceptual model and have no attached implementation. To symbolize data exchange in form of messages, the conceptual icons are connected using several framework-specific connectors. It is important to highlight that the conceptual model does not have an attached implementation, and represents simply an initial drawing of the model that the composer wants to develop. Once all icons are connected, the conceptual model is ready for syntactic verification. The framework creates a production string that is constructed by linear arrangement of icon types and their adjoined connectors. The production string is then submitted to a composition grammar parser, which can accept or reject it. If the grammar parser accepts the production string, then the conceptual model is considered syntactically correct and the next stage, model discovery and selection, can proceed. In contrast to current approaches, we propose the syntactic verification of the conceptual model instead of the syntactic verification of the discovered composed model because the latter is a costly process. This is facilitated by our approach to data encapsulation, component connection, and the representation of the composed conceptual model.

As discussed above, the main issues towards model composition and verification

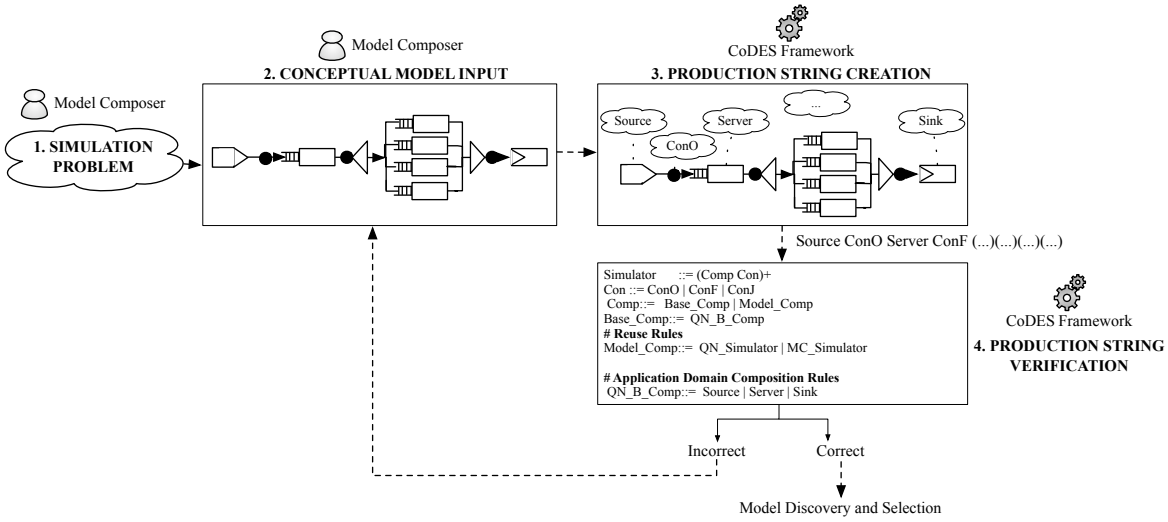


Figure 4.2: Overview of Syntactic Composability and Verification

are data encapsulation, e.g. how components are separated from each other, component connection, e.g. how data is exchanged between components, the representation of the composition, and the verification of syntactic composability. In Chapter 3, we presented how the CoDES framework effortlessly permits component isolation and loose coupling through the *black-box component-connector paradigm* [118]. 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 of type *one-to-one* (Connect) 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. These are shown in Figure 3.3.

Several design considerations have led to the adoption of a component-connector paradigm, in which communication between components is done only through *connectors*. Firstly, by using a component-connector paradigm together with a black-box component representation, heterogeneous components need not consider all possible connection scenarios with other components. Instead, the focus is on proper com-

munication with a reduced number of well-specified connectors. Next, to ensure that components communicate in a standard way, messages that are passed by connectors between components must adhere to the framework standard COML schema. This guarantees that all components communicate in a standardized, error-free (with respect to the language syntax) format that they are able to understand. Lastly, to facilitate easy re-combination of components, each component must implement its own time management mechanism. Regardless of each particular component time management approach, all inter-component messages are delivered by the CoDES connectors in a timely fashion. The CoDES connectors time-stamp each message with the current framework time and guarantee FIFO delivery in an unbounded queue to each component. Figure 4.3 illustrates the CoDES component-connector paradigm for a simple single-server queue example using the base component icons presented in Figure 3.3.

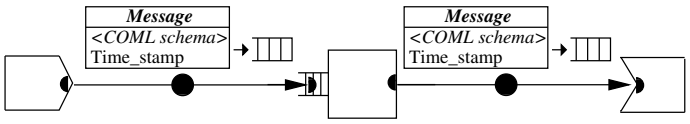


Figure 4.3: Conceptual Model for a Single-Server Queueing System

4.2.1 Compositional Grammars

As shown in Figure 4.2, we propose to convert a conceptual composed model from a graphical representation into a production string for automated verification of syntactic composability. The production string is verified by a composition grammar, which specifies general and specific connection rules of components in the CoDES framework, within and across application domains. Towards this, we propose to specify model composition and reuse rules using regular grammars [54]. This formalizes syntactic composability and supports scalable syntactic verification of the composed conceptual model because the process translates to fast acceptance verification of the production

string by a composition grammar parser [37]. Moreover, the production string that describes the composed model can be further used to formulate search criteria to discover plausible models and model components. For example, the production string of the query model can be hashed as a key in a DHT-based overlay network and the discovery process translated into a fast lookup query [9].

Figure 4.4 shows the CoDES composition and reuse grammar in Extended Backus-Naur Form (EBNF) [57].

```
# CoDES Composition Rules
# Across Application Domain
Simulator ::= (Comp Con)+
Con       ::= ConO | ConF | ConJ
Comp      ::= Base_Comp | Model_Comp
Base_Comp ::= QN_B_Comp | MS_B_Comp | ...
# Reuse Rules
Model_Comp::= QN_Simulator | MC_Simulator

# Application Domain Composition Rules
QN_B_Comp  ::= Source | Server | Sink
...
```

Figure 4.4: CoDES Composition Grammar

The composition grammar is organized in two levels. The first level defines the rules for the composition of models across application domains in the CoDES framework. The second level defines application-domain specific composition rules that describe the composition within an application domain. Informally, the CoDES composition rules specify that components are interconnected through connectors, according to the component-connector paradigm. A CoDES simulator consists of a set of model components *(Comp)* interconnected by connectors *(Con)*. *Comp* is a base CoDES component *(Base_Comp)* selected from an application domain such as queueing networks *(QN_B_Comp)*, or a reused component *(Model_Comp)*, from the CoDES component repository. Components are interconnected by connectors *(Con)* such as one-to-one *(ConO)*, fork or one-to-many *(ConF)* and join or many-to-one *(ConJ)*. Reused model components can be developed from reused simulators saved in the repository

as components either containing only components from a specific application domain *(QN_Simulator)* or from across application domains *(MC_Simulator)*. Section 4.3.2 presents an example of the composition grammar for the Queueing Networks application domain. The inherent benefit of a composition grammar is that it enhances the automated verification of syntactic composability as discussed above.

4.2.2 Verification of the Conceptual Simulation Model

Syntactic composability verification is the process of checking that the components in the composed model interoperate properly and that the composition structure is in accordance with framework and application domain rules. Proper interoperation refers to correct connection, compatible data exchange and correct message passing [88, 30]. In current approaches [61, 133], the verification of syntactic composability is mostly done at source code level. This implies that component discovery must be performed before syntactic composability verification. However, model discovery and selection is a costly process, which in this case will be performed regardless of the correctness of the component connections and their potential incompatibilities. An evaluation of the cost of model discovery and selection is presented in Chapter 8. We propose to verify syntactic composability before model discovery and selection. In other words, we propose to verify the syntactic composability of the conceptual composed model *before* the conceptual icons have an attached implementation.

Towards this, the CoDES framework enforces standards for component communication and connection to facilitate the switch in the order of life-cycle steps. Specifically, our proposed framework guarantees that: (i) the number of communication channels of the component implementation is equal to the number of channels of the icon representation, and (ii) components and their communication are represented consistently by the same schema. From (i) it follows that the component connection will be the same

regardless of whether the components are only conceptual icons or have attached implementations. Next, (ii) guarantees that all repository components use the same standardized syntax for communication. As such, we guarantee that any implementation discovered for each of the conceptual icons in the conceptual model will be properly connected and compatible with the rest of the discovered components, provided that the conceptual icons are properly connected and the composition structure is correct. Thus, the process of syntactic composability verification translates to checking that the components are properly connected and that the composition structure adheres to the rules specified in the CoDES composition grammar. Chapter 8 presents an evaluation of the cost of model discovery and selection and highlights the inherent advantages of our approach.

Figure 4.5 presents the pseudo-code for the syntactic composability verification algorithm. In line 1, the algorithm first verifies that all component icons are properly

```
boolean syntacticVerification(Composition comp){
1.      if (correctlyConnected(comp)){
2.         String prodString = constructProductionString(comp);
3.         Parser parser = new Parser(getDomainGrammar());
4.         return parser.parse(prodString);}
5.      return false;
}
```

Figure 4.5: Pseudo-code for Syntactic Composability Verification

connected. If components are properly connected, a production string that defines the composition is obtained on line 2. The production string is formed according to the linear arrangement of the component icons on the drawing panel. A new Earley parser is created using the composition grammar specific to the application domain on line 3. Next, if the parser accepts the production string on line 4, the composition is syntactically correct. Otherwise, the composition is not syntactically correct and the component-based life-cycle cannot proceed to the model discovery and selection stage.

4.3 Theoretical and Experimental Analysis

In current approaches, the cost of syntactic composability verification is a function of the cost of compiling the source code of the composed model, which in turn is a function of the size of the source code, as well as the compiler and its inner structure. The size of the source code is a function of the number of components, and the number of attributes per component, the number of connectors in the composition. In contrast, the cost of syntactic verification in our proposed approach is a function of the number of components and the number of connectors, as shown below. This is because in our proposed approach we do not compile the composition source code, but verify a production string representing the composition against a composition grammar. We evaluate this cost using theoretical and experimental analysis.

4.3.1 Theoretical Analysis

In syntactic verification, we employ a grammar parser to determine if the composition production string is accepted by the CoDES composition grammar. This is implemented using Earley's parsing algorithm, a top-down dynamic programming algorithm for parsing context-free languages [37]. The algorithm progressively goes through the production string and constructs an ordered state set with all possible rules that can be executed for each symbol in the production string.

The state set contains tuples in the form $(X \rightarrow \alpha \bullet \beta, i)$, where $(X \rightarrow \alpha\beta)$ is the current production rule being matched, $\bullet$ represents the current position in the rule, and i represents the position in the input where the current matching began (at first i represents the initial position). Initially, the state set contains $S(0)$ which contains the top-level rule in the grammar. The parser then iteratively operates in three stages, namely *prediction* in which the parser attempts to predict the next rule to be fired,

scanning, in which the parser adds new rules based on the next symbol in the input stream, and *completion* in which the prediction is matched with the input stream.

In the general case, the complexity of the Earley parser is $O(l^3)$, where l is the length of the string submitted for parsing. As such, regardless of the type of each application domain composition grammar, in the general case the complexity of the syntactic composability verification step, O_{syn} is $O_{syn} = O(l^3)$ where l, the length of the string, depends on the number of components n and the number of connectors c (and their additional representation, such as the "(" symbol)): $l = n + 3 * c$. However, the number of connectors c is a polynomial of the number of components, and as such we have:

$$O_{syn} = O(l^3) = O((n + 3 * P^1(n))^3) = O(n^3)$$

The complexity of the Earley algorithm is greatly improved for different types of composition grammars. For example, for an unambiguous grammar the complexity would be $O(n^2)$, whereas for most deterministic context-free grammars such as the queueing networks composition grammar in Figure 4.6, the complexity is $O(n)$ [37].

4.3.2 Experimental Analysis

Example of Queueing Networks Applications

Extended Composition Grammar

An application domain composition grammar contains composition rules using application domain-specific base components and is added to the CoDES grammar whenever a new application domain is added to the framework. As shown in Figure 3.3, base components in the Queueing Networks application include 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*.

Composition rules specific to the Queueing Network Application Domain define the connectivity of the basic components to form different queueing network systems using the three types of CoDES connectors presented in Figure 3.3. Figure 4.6 presents the queueing networks application domain composition grammar.

```
# Application Domain Composition Rules
# 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 | Model_Comp
```

Figure 4.6: Composition Grammar for Queueing Networks Application Domain

To allow for component reuse, a queueing network simulator can include base components such as a server *(Server)* or model components from the repository *(Model_Comp)*. The composition rules allow for the composition of a plethora of open, closed and hybrid queueing networks. The composed simulator can be in turn placed in *Model_Comp* as a model component for reuse.

Figure 4.7 presents a simple single-server queue example as it was created by the model composer using the CoDES GUI.

After the model composer draws the conceptual model, syntactic composability is

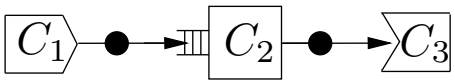


Figure 4.7: Simple Single-Server Queue

verified. As discussed in Section 4.2.2, the module first checks that no component port is left unconnected. Next, a production string is generated to describe the conceptual model. The production string is a linear arrangement of the component types according to their position on the graphical screen. The production string for this model is:

$$Simple\_QN = Source\ ConO\ Server\ ConO\ Sink$$

The model is syntactically verified if the production string is accepted by the Queueing Networks composition grammar. For this example, the algorithm in Figure 4.5 returns a positive answer.

As an initial evaluation of our proposed syntactic composability verification process, we perform syntactic composability verification on a single-server queue model with varying number of components (e.g. Server units) and present our results in Table 4.1. We analyze models with 10, 100, 500, 1,000, and 10,000 components, in which the Server units are connected serially, one after another, and the Source and Sink components are connected similarly to Figure 4.7. The experiments are conducted on a machine with Intel Core 2 Duo CPU E6550 @ 2.33 GHz processor and 4 GB SDRAM, running Ubuntu Linux 8.04 (64 bit). The results in Table 4.1 represent an average of 50 execution runs. The runtime of the syntactic composability verification algorithm varies in accordance with the theoretical evaluation, however, the runtime is promisingly small. For example, for a single-server queue model with 500 components, the runtime is around one tenth of a second on a commodity desktop PC. For 10,000 components, the runtime is less than 6 seconds on a commodity PC. These results are en-

# Components	Runtime ($s * 10^{-3}$)
10	4.60
100	15.82
500	111.94
1,000	802.07
10,000	5,746.45

Table 4.1: Syntactic Composability Verification of a Single-Server Queue Model

couraging and show that our approach for syntactic composability verification is highly scalable.

Experiment on Automated Model Generation

An important issue that prevents the wide acceptance of component-based development paradigms is the lack of a component repository where components can be shared [31, 61]. One of the major advantages of the CoDES compositional grammar is that syntactically correct simulators can be easily built in the CoDES framework, using the grammar to generate correct models of a given size. We generate a repository of syntactically correct models with up to 10 components using the CoDES composition grammars. These syntactically correct generated models can be enhanced to become semantically correct generated models by adding component implementations that are semantically compatible, as shown in Chapter 6. Using this approach, an initial repository can be easily developed for each application domain that is added to the CoDES repository. Furthermore, the CoDES framework can be used straightaway, without the delay necessary for the component repository to reach useful size and diversity.

Consider the Queueing Networks application domain and a composition grammar that describes open queueing systems with one Source and one Sink. Using this grammar, we can generate a repository of syntactically valid simulators with only base components, with a given number of components. These simulators are *syntactically correct* models. The generation process is bounded by the number of components. A recursive

algorithm parses the grammar rules until the desired number of components is reached and the rules reach a terminal symbol. Table 4.2 presents the number of syntactically valid simulators with the total number of components ranging from 4 to 10, and the time taken to generate the models. The time taken to generate the models is an average of 10 runs. The generation was executed on the same machine as the previous experiment.

# Components	**# Syntactically Correct Models**	**Runtime** ($s * 10^{-3}$)
4	2	8.8
5	6	23.0
6	19	63.5
7	57	182.8
8	164	442.0
9	457	1,392.0
10	1,244	4,626.3
Total	1,949	6,738.4

Table 4.2: Number of Generated Models in the Repository

Using a simple composition grammar, we are able to generate 1,949 syntactically correct models, with a total number of components of up to 10, in less than 7 seconds. These models can become semantically valid models through the addition of semantically compatible component implementations, as shown in Chapter 6. As such, we are able to obtain a fairly large initial repository of model components and standalone simulators at insignificant costs, resulting in a valuable testbed for the validation of the CoDES framework.

4.4 Summary

Towards model composition, we propose the component-connector paradigm in which the component implementation is hidden from other components in the framework and components are linked through framework-specific connectors. The implementation of the component-connector paradigm, in which components are black-boxes with in

and/or out communication channels interconnected through connectors, guarantees that components implement communication stubs only with standardized connectors. The communication between components is done through messages standardized in our proposed COML format. Thus, components not only interoperate with each other, but they can be recomposed to form new models to suit specific user needs. This captures the true meaning of syntactic composability as discussed in [30]. We add a semantic meaning to syntactic composability by describing compositions through production strings validated by composition grammars. To the best of our knowledge, this is a novel approach in which the framework itself can provide for various application domains.

In contrast with current approaches, we verify syntactic composability *before* the costly model discovery and selection. To facilitate syntactic composability verification, we describe connection rules within and across application domains using the CoDES composition grammar expressed in EBNF. This provides a more structured approach to syntactic verification of composed models and also offers extendability to include new application domains. To provide for scalability, the composition grammar is organized in two levels. The first level presents the overall composition rules that characterize the entire framework. These are an implementation of the component-connector paradigm. The second layer contains all application domain-specific composition rules, which describe the application domain base components and their permitted connections. A composition grammar to describe syntactic composability also facilitates model discovery by providing a linear production string to syntactically describe the composition. This provides a simple and efficient method of searching plausible models in the repository to support reuse. Our experiments show that our approach is highly scalable, with models of 500 components verified in less than one second, and models of 10,000 components verified in less than 6 seconds, on a commodity desktop PC.

Chapter 5

Model Discovery and Selection

Given a syntactically correct composition of conceptual components, component discovery and selection is performed to identify components with an attached implementation that match the user query or objective. Discovery can be performed in three ways, namely: (i) the model composer specifies only a *composition objective*; (ii) the model composer draws the conceptual model and specifies *queries for individual components* or parts of the composition; and (iii) the model composer draws the conceptual model and the *discovery of simulators* with the same or similar structure is performed. The first type of discovery and selection does not require the construction of any conceptual model or syntactic composability verification. However, it is an NP Complete problem [13, 85]. Additionally, the composition objective is difficult to specify, and as such heuristics to by-pass the NP Complete problem are hard to implement in a real system. Accordingly, we propose the second and third type for model discovery and selection, namely, the discovery of individual components, and the discovery of entire simulators.

Several factors influence model discovery and selection. Firstly, the component query and knowledge about components must be represented in an appropriate format to facilitate meaningful discovery. Towards this, ontologies have been proposed in software engineering to represent knowledge about components and domains in gen-

eral [15, 83]. In modeling and simulation, ontologies have been proposed to represent knowledge about application domains [77, 109], but, to the best of our knowledge, there does not exist a simulation ontology that captures knowledge about application domains and components with detailed definitions and implementations. In contrast, component-based software engineering methods are more detailed and automated, but focus only on basic programming language constructs such as functions. Secondly, exact matches between the user query and the repository component are very rare. As such, means to *rank partial matches* and select components in the repository based on the user query are needed [61]. Adequate methods for ranking and selecting are necessary especially in the context of a large component repository. Thirdly, the size of the component repository, as well as the number of attributes per component drastically, influences the performance of the discovery and selection service. In this chapter, we present our approach to component discovery and selection using our proposed component-based ontology to meaningfully rank repository components according to a user query. For ease of reading, we exemplify our approach using a single-server queue example. More complex and detailed examples are presented in Appendix B and Appendix C.

5.1 Current Approaches

Current work that employs the Base Object Model, presented in Chapter 2, proposes a layered approach to simulation composition [75]. Relevant to component discovery and selection is their proposed BOM Matching and Composition phase, which attempts to discover the appropriate BOMs for a specific simulation scenario. BOM Composition attempts to find the best composition of discovered BOMs that suits the scenario. Model discovery is performed based on component syntax, i.e., event signature, and component semantics, i.e., the input and output of the component. In this stage, partial matches are not ranked. For each component query there might be a set of candidate

components that fit the specified scenario. All possible combinations of candidate components are executed and compared with the simulation scenario. If the order of event calls is the same or similar, the discovery process is successful. An important point to highlight is that this approach requires a low level detailed scenario that might not be easily available. Furthermore, there is no means to rank the discovered candidate components, which would reduce the computational complexity of the process.

Recent work that uses the DEVS formalism, CellDEVS++ (CD++) [24], proposes an online repository of CD DEVS models together with their experimental frames. The repository stores atomic and coupled DEVS models using a tree structure to describe model coupling. For each model, the experimental frame contains input data and the set of experiments that can be performed with the model. Experiment data includes a textual description of the experiment assumptions and constraints. On top of the experiment data, the online repository also saves all the experiment results. Model discovery is performed based on data from the experimental frames, including information about how the model can be used. However, there are no means yet of ranking partial matches based on a query. Furthermore, the search process compares only string text values without attached semantics and in the absence of linguistic algorithms to determine similarities.

In general work that looks only at model discovery and selection, Aronson and Bose [6] describe the simulation model query and selection process with respect to a user specified simulation scenario. A quality function is proposed as a possible solution to partially ordering discovered simulation models, but details are not provided and no follow-up work exists. General work that focuses on the use of ontologies for simulation modeling is that of Silver et al. [109, 108]. The focus of their work is on building semantically aware simulations based on domain specific ontologies. They propose the Ontology Driven Simulation (ODS) design tool that maps concepts in the domain on-

tology (e.g. healthcare) to simulation concepts in their proposed DEMO ontology [71]. The DeMO project proposes a technique for transforming models from various application domain ontologies, e.g. glycan biosynthesis, into the DeMO ontology and subsequently into an executable simulation. This is an important step in representing simulation domain knowledge in a format readable by computers and humans alike. Another example is the use of ontologies in agent-based simulations [99]. Here, an ontology is proposed for the representation of key modeling concepts such as conceptual model, communicative model, programmed model, and experimental model. Several venues for automated reasoning are highlighted, including inferring assumptions, inferring parameters and assumptions, and validation. In this theoretic proposal, the ontology would be the center of the component-based framework, with all composition and reasoning performed on it. This would be feasible if the ontology contained complete information about the composition, components, and application domains, which is rarely the case. In contrast, in our approach the ontology is a secondary tool in the composition process, used for domain knowledge representation. Our component abstraction uses meta-data that is described in our ontology, but also relies on a framework-specific standard. For syntactic composition, we propose composition grammars, which are better suited to represent composition structure. Lastly, we propose a well-defined process for model discovery and selection as well as semantic composability validation, which relies only partly on the ontology.

In software engineering, Tansalarak and Claypool [124] propose a combination of different matching techniques to provide a ranked set of highly qualified software components from the repository. However, no measure of the correctness of the composition is discussed. Furthermore, as is generally the case for software engineering, the focus is on the reuse of program code such as functions, an approach not suited for the discovery of simulation components.

5.2 Proposed Approach

Among the three types of model discovery discussed above, we propose the last two. Specifically, once the conceptual model is drawn and syntactically verified, the model composer can discover: (i) individual base or model components based on a user query; (ii) parts of the conceptual model in the form of model components; and (iii) the entire simulator. The discovery of the entire simulator can be easily implemented as a lookup query of a hashed production string that describes the simulator over a DHT overlay with nodes representing all the standalone simulators in the repository. In this chapter, we focus on the discovery and selection of individual components, namely base components. The discovery of individual base components provides the foundations of the more complex discovery of model components. Model components represent a more difficult problem because they are composed of other base and/or model components. Since the discovery of a model component cannot be implemented as the iterative discovery of all its base components, different abstractions must be in place. We propose a simple solution to this problem in this chapter and discuss several abstraction trade-offs in Chapter 9 as directions for future work.

Base components are discovered one by one, based on the user query. Once all base components are discovered, the entire composition is semantically validated. The pseudo-code for our discovery and selection algorithm is shown in Figure 5.1. For each component in the repository, if the component is not eliminated based on our elimination strategy to reduce the search space (line 4), the matching index is calculated to measure the component similarity between the query component and the repository component (line 6). The Matching Index is defined in Section 5.2.2. A list containing the ordered set of repository components and their matching index result is created and returned.

```
void discoveryAndSelection (MetaComponent query,
                        ArrayList<MetaComponent> repository,
                        Composition composition, String ontFile){
1.  ArrayList<MetaComponent> result = new ArrayList<MetaComponent>();
2.  HashMap<double, MetaComponent> aux =
                        new HashMap<double, MetaComponent>();
3.  for (MetaComponent comp :  repository){
4.      if (eliminated(comp, query, composition)) continue;
5.      reasoner = createReasoner(OWL_REASONER,ontFile);
6.      double MI = calculateMI(query, comp, reasoner);
7.      addToHashmap(MI, comp);
8.  }
9.  sortHashmapOnMI(aux);
}
```

Figure 5.1: Pseudo-code for Model Discovery and Selection

As we have seen above, adequate knowledge representation is required to describe the component query and domain information for meaningful discovery. An ontology provides the best means to represent knowledge and relations between entities in a format readable by humans and computers alike [15]. We propose COSMO (COmponent Simulation and Modeling Ontology) to describe component-based simulation within and across application domains to suit the multi-level application domain perspective in the CoDES framework [125].

On the other hand, similarity is not an exact answer and as such partial matches need to be evaluated. We propose the Matching Index to rank partial matches of base components in the repository, based on component attributes and behavior (line 6). The Matching Index uses knowledge in the COSMO ontology to calculate the degree of similarity between the query and repository component attributes and behavior.

Lastly, the size of the component repository and the average number of attributes per component drastically influence the performance of the discovery and selection service. We propose simple elimination heuristics that employ the structure of the conceptual model to discard parts of thse search space (line 4). This is an important elimination, since, as we have seen in the previous chapter, the start-up size of the CoDES repository is around 2,000 components.

5.2.1 Domain Knowledge Representation

COSMO semantically enriches the description of model components to support model discovery and selection. Using COSMO, information about components and application domains can be organized in a structure that facilitates discovery. For example, when searching for *Server* components, a hierarchy such as {*SingleUnitServer* `subClassOf` *Server*}; {*OpenSource* `subClassOf` *Source*}, will give a higher ranking to components of type *SingleUnitServer*, than to components of type *OpenSource*.

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 shown in Figure 5.2. To achieve generality across applica-

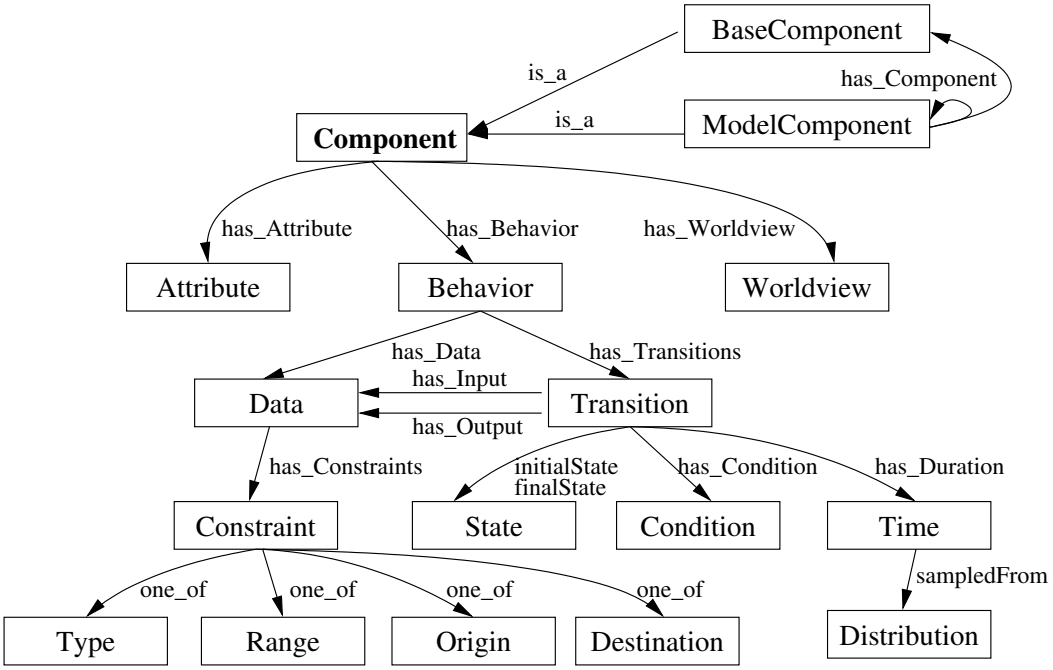


Figure 5.2: COSMO Ontology Structure

tion domains and at the same time support specific application domain requirements, we include first an application domain oriented component hierarchy. As discussed in Chapter 3, the component repository consists of shared components common to all

application domains termed as *model components*, and shared fundamental entities specific to each application domain, termed *base components*. Thus, an application domain defines its own specific pool of *BaseComponents*. Also, composed models are placed in the component repository as *ModelComponents*. The second set of classes describes 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 by 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.

The COSMO Ontology is written in OWL DL [83, 8] and has been developed using Protégé [92], a widely used ontology builder. Figure A.3 presents a snapshot of the asserted COSMO class structure in Protégé.

5.2.2 Measure of Component Similarity

Model discovery and selection is the process of ranking repository components based on the simulator developer query, in order to locate implementations for the base and model components in the conceptual model. In our proposed approach each component can be discovered *individually* based on a user specified query. Alternatively, the *entire* composition can be discovered based on the conceptual model structure and other user queries. The latter will search for previously validated simulators that have been saved into the repository for reuse "as-is". In this chapter, we focus on the discovery of individual base components. Once all components have been discovered, the composition is integrated and semantically validated. If the composition is not valid, then

a new iteration of component discovery can be performed. In general, there will not exist an exact match between the query component and the repository components. As such, a method to *rank* the repository components and *select* only relevant candidates for composition is needed.

We propose the *Matching Index* (MI) to meaningfully rank the repository components based on the developer query. MI encapsulates the syntactic and semantic relevance of the repository components with respect to the query component. In this context, *syntactic information* refers to component mandatory attributes, production string and structure (for model components). *Semantic information* refers to the component behavior in terms of states and input and output transformations.

The Matching Index considers attribute names and values, as well as component input and output transformations, as shown in 5.3. The calculation of MI considers

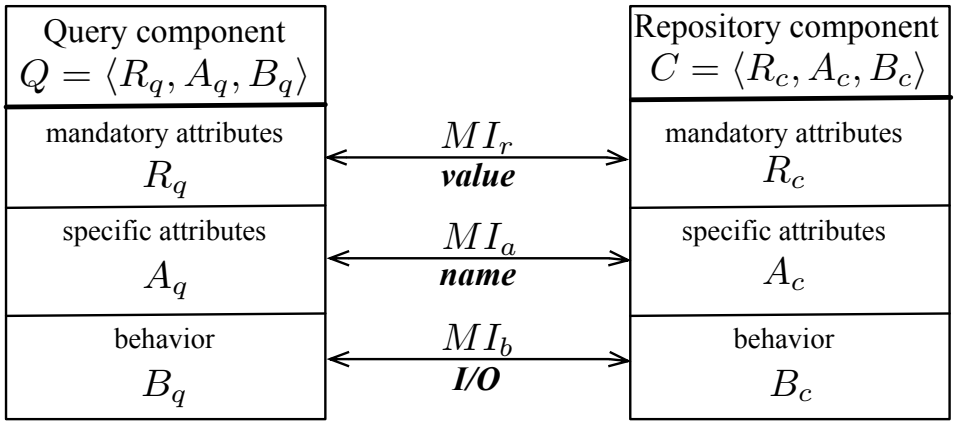


Figure 5.3: Semantic Ranking using Matching Index

attribute *values* for the mandatory attributes, and attribute *names* for the specific attributes. Since specific attribute values can be modified before simulation execution, we are interested in repository components that can be described using the same or similar specific attributes. On the other hand, for mandatory attributes such as `author`, `location`, etc., that will not be modified by the simulation developer, we are interested in repository components with similar or identical values.

Definition 6 (Matching Index). Let $Q = \langle R_q, A_q, B_q \rangle$ denote a query component, and $C = \langle R_c, A_c, B_c \rangle$ represent a repository component. The matching index $MI(Q, C)$ is defined as

$$MI(Q, C) = w_r * MI_r(R_q, R_c) + w_a * MI_a(A_q, A_c) + w_b * MI_b(B_q, B_c) \quad (5.1)$$

where $MI_r, MI_a, MI_b \in [0, 1]$ are matching indexes for the required attributes, component attributes and behavior respectively, and $w_r, w_a, w_b \in [0, 1]$ are the respective index weights.

The Matching Index is expressed as the weighted sum of the partial matching indexes between a query component and the repository component. The partial matching indexes refer to the similarity between the required attributes (MI_r), the component attributes (MI_a), and the behavior (MI_b) of the two components. The values for MI_r, MI_a, and MI_b are defined below.

Definition 7 (Required Attributes Matching Index). Let R_q and R_c be the set of required attributes for the query component Q and the repository component C respectively, where $R = \{r | r = (name, value)\}$. The matching index MI_r of the required attributes is defined as

$$MI_r(R_q, R_c) = \frac{\sum_{r_q \in R_q} m(r_q, r_c)}{|R_q|} \quad (5.2)$$

where $r_c \in R_c$, with $name(r_q) = name(r_c)$, and $m(r_q, r_c)$ is the matching function between the attributes defined as

$$m(r_q, r_c) = \begin{cases} 1 & \text{if value}(r_q) = \text{value}(r_c) \\ 0 & \text{otherwise} \end{cases}$$

The required attribute matching index determines how many of the required attributes of the repository component are *the same* with the query component's required attributes values. Even though the matching function computes exact matches, component descriptions can be evaluated for relevant similarities using linguistic algorithms.

Definition 8 (Component Attributes Matching Index). Let A_q and A_c be the set of component attributes for the query component Q and the repository component C respectively, where $A = \{a|a = (name, value)\}$. The matching index MI_a between the required attributes is defined as

$$MI_a(A_q, A_c) = \frac{\sum_{a_q \in A_q, a_c \in A_c} m(a_q, a_c)}{|A_q|} \tag{5.3}$$

where $m(a_q, a_c)$ is the matching function between the specific component attributes defined as

$$m(a_q, a_c) = \begin{cases} 1 & \text{if name}(a_q) = \text{name}(a_c) \\ 0.75 & \text{if name}(a_q) = \text{subPropertyOf(name}(a_c)) \\ 0.5 & \text{if subPropertyOf(name}(a_q)) = \text{name}(a_c) \\ 0 & \text{otherwise} \end{cases}$$

In the case of component attributes, it is important to determine if a repository component can be described using the same or similar attributes. Thus matching is done with respect to the *name* of the attributes and not to the value as previously. The attribute matching index has also a semantic dimension, in which attributes are compared based on their position in the COSMO ontology. In the case where a_c, the repository component's attribute, is a *subProperty* of a_q, that is a_c is more "specialized" than a_q, then the matching index is less (i.e., 0.5) than when a_q is more specialized than a_c (i.e.,

0.75). The latter situation is more desirable than the former, hence the greater value for $m(a_q, a_c)$, $m(a_q, a_c) = 0.75$, than for the former where $m(a_q, a_c) = 0.5$. Table 5.1 presents examples to showcase the different values of $m(a_q, a_c)$.

Query Component Attribute	Repository Component Attribute	$m(a_q, a_c)$	Comments
$(interArrivalTime, 5)$	$(interArrivalTime, 20)$	1	Component attributes have the same name. Value not important as it can be changed by the model composer.
$(interArrivalTime, 5)$	$(waitingTime, 40)$	0.75	$interArrivalTime$ is more specialized than '$waitingTime$
$(interArrivalTime, 5)$	$(interArrivalExponentialTime, 3.4)$	0.5	$interArrivalExponentialTime$ is too specialized.
$(interArrivalTime, 2.5)$	$(noUnits, 4)$	0	No relation.

Table 5.1: Example of Component Attributes Matching Index Calculation

The behavior matching index MI_b considers the constraints on the input and output data of the two components, since this is the basic information the user could provide. In an ideal situation in which the user is willing to provide more complete information about the query component, MI_b can be extended to include measures of components state machine similarity, conditions on attributes, etc.

Definition 9 (Behavior Matching Index). Let B_q and B_c be the behavior of the query component Q and the repository component C respectively. Let IC_q, OC_q, IC_c, and OC_c be the set of constraints on the input and output data for the query and repository component, where a set of constraints is defined as $C = \{c = (type, value) | type \in \{range, origin, destination, class\}\}$ is a constraint set. The behavior matching index MI_b is defined as

$$MI_b(B_q, B_c) = \frac{\sum_{c_q \in IC_q, c_c \in IC_c} m(c_q, c_c) + \sum_{c_q \in OC_q, c_c \in OC_c} m(c_q, c_c)}{|IC_q| + |OC_q|} \tag{5.4}$$

where $type(c_q) = type(c_c)$, and $m(c_q, c_c)$ is the constraint matching function de-

fined as

$$m(c_q, c_c) = \begin{cases} 1 & \text{if value}(c_q) = \text{value}(c_c) \text{ and constraint\_type}(c_q) = \text{constraint\_type}(c_c) \\ 0.75 & \text{if typeOf(value}(c_c)) = \text{value}(c_q) \text{ and constraint\_type}(c_q) = \text{constraint\_type}(c_c) \\ & \text{and constraint\_type}(c_q) = (\text{origin} \vee \text{destination}) \\ 0.75 & \text{if value}(c_c) \subseteq \text{value}(c_q) \text{ and constraint\_type}(c_q) = \text{constraint\_type}(c_c) = \text{range} \\ 0.75 & \text{if type(value}(c_c)) \subseteq \text{value}(c_q) \text{ and constraint\_type}(c_q) = \\ & \text{constraint\_type}(c_c) = (\text{type} \vee \text{class})) \\ 0.5 & \text{if typeOf(value}(c_q)) = \text{value}(c_c) \text{ and constraint\_type}(c_q) = \text{type}(c_c) \\ & \text{and constraint\_type}(c_q) = (\text{origin} \vee \text{destination}) \\ 0 & \text{otherwise} \end{cases}$$

MI_b encapsulates semantic information by comparing the relations between the constraints of *type, range, destination,* and *class*. As discussed in detail in Chapter 3, data constraints are used to describe a simulation component from an external perspective, with respect to the data it can exchange with its neighbors. When no exact match is found, MI_b helps to identify repository components that can substitute the query component with respect to input and output data. This is done by evaluating the value of the *typeOf* and *subClassOf* predicates in the COSMO ontology. $m(c_q, c_c)$ returns a higher value (0.75 or 1) when c_q subsumes c_c (c_c is stricter than c_q and C is a "specialized" version of Q) and lower values (0 or 0.5) otherwise. For the constraints of type range, $m(c_q, c_c)$ returns higher values if the interval of c_c is included in the interval for c_q and lower values otherwise. Similarly, the differences in the values for $m(c_q, c_c)$ signify that a particular case is more desirable than another. Table 5.2 presents an example of the computation of $m(c_q, c_c)$.

Query Data Constraint	Repository Data Constraint	$m(c_q, c_c)$	Comments
$(range, [3.5, 5.9])$	$(range, [3.5, 5.9])$	1	Identical types and values.
$(origin, Server)$	$(origin, SingleUnitServer)$	0.75	Identical constraint types, of kind "origin". Repository component data is more specialized.
$(range, [3.5, 11.5])$	$(range, [4.5, 7.0])$	0.75	Identical constraint types, of kind "range". Repository component range included in the query.
$(type, Job)$	$(type, CustomerJob)$	0.75	Identical constraint type, of kind "class". Repository component has a more specialized data type.
$(type, double)$	$(type, int)$	0.75	Identical constraint type, of kind "type". Repository component has a more specialized data type.
$(destination, SingleUnitServer)$	$(destination, Server)$	0.5	Identical constraint type, of kind "destination". Query requires more specialized data.
$(destination, SingleUnitServer)$	$(range, [4.5, 7.0])$	0	No relation.

Table 5.2: Example of Behavior Matching Index Calculation

Discovery of Model Components

As discussed in Chapter 3, model components are obtained by stripping off some components from a semantically valid standalone simulator at specific cut-off points, such that the resulting model component has both "in" and "out" communication channels. For the ranking of such model components based on a user query, we propose to look at mandatory attributes, and at input/output data that is received/sent by the component(s) at cut-off points. Then the formula for the matching index for model components becomes:

$$MI_m(Q, C) = w_r * MI_r(R_q, R_c) + w_b * MI_b(B_q, B_c)$$

As it can be seen, the difference between MI_m and the proposed MI, is the absence of the specific attribute matching index. This is because there is no information specific to the model component that is saved into the repository. The model component information that is saved with the model component is also shown in Table 3.1.

Optimizations to Reduce the Search Space

The size of the component repository is an important factor in the runtime of the discovery service, because the Matching Index is calculated for every query request-

component pair. We perform several optimizations to reduce the number of repository components considered in the discovery of both base and model components. In both cases, the conceptual model represents an additional aid in the discovery process by providing information about component types and the neighbors with which they can communicate.

When *discovery of base components* is performed, we consider only repository components of the same *type* as the base components in the conceptual model. For example, when discovery is performed for component C_2 in Figure 4.7, only components of type $Server$ (or additional sub-types according to the COSMO ontology) are considered. In this particular case, the CoDES component repository contains `1,961` base and model components for the Queueing Networks application domain, from which `1,949` are model components that contain between 4 and 10 base components connected in different ways, and 9 are base components, 3 each for the types *Source*, *Server*, and *Sink* respectively. This repository models a real-life scenario where we expect the number of base components per application domain to be significantly smaller than the number of developed component-based models. Thus, when a query is performed for component C_2, the number of candidate components is reduced to 3, which represents a decrease of `99.99%`.

When *discovery of model components* is performed, we reduce the search state space by considering only the repository model components whose input and output constraints match (according to the COSMO ontology) those that can be deduced from the conceptual model. For example, for the model component in Figure 8.10 (b), we only consider model components that receive input from a *Server* type base component, and send output to a *Sink* component. An evaluation of the improvements of this optimization is presented in Chapter 8.

In the current implementation of the discovery service, an ordered set of components

with non-zero MI is returned for each query. To reduce the size of this set and to facilitate calculations of precision and recall measures [21], it would be appropriate to return only the components with a non-zero MI higher than a given *threshold*. However, the threshold value depends on the amount and specificity of the query information and the size and content of the component repository and is not within the scope of our study.

5.3 Theoretical and Experimental Analysis

The runtime cost of model discovery and selection is a function of the repository size, i.e., the number of components in the repository, the number of attributes in the query, and the number of attributes per component in the repository. In this section, we will show that the ontology and the reasoner employed for querying also influence the discovery cost. The current CoDES repository contains `1961` components, from two application domains, namely Queueing Networks and Military Training Simulations. The Queueing Networks application domain contains `1958` base and model components. The newly added Military Training simulations application domain contains for now only three components, two base components and a simulator. We perform two experiments. Firstly, we showcase how the Matching Index is calculated using two repository components from the Queueing Network application domain. Secondly, we evaluate the runtime for the two queries. As before, the experiments were executed on a machine with Intel Core 2 Duo CPU E6550 @ 2.33 GHz processor and 4 GB SDRAM, running Ubuntu Linux 8.04 (64 bit).

5.3.1 Theoretical Analysis

For each component in the composition, we calculate the matching index MI between the simulator developer query $Q = \langle R_q, A_q, B_q \rangle$, and each component in the repository

$R = \langle R_c, A_c, B_c \rangle$. The process employs the COSMO ontology to determine the similarity between component attributes and behavior as discussed above. Thus, the time complexity of the discovery process, O_{disc} becomes:

$$O_{disc} = n * r * (|R_q| * |R_r| + |A_q| * |A_r|) * O_{ontology}$$

where n is the number of components in the conceptual model, r is the number of candidate repository components, $|R_q|$ and $|R_r|$ are the number of mandatory attributes, $|A_q|$ and $|A_r|$ are the average number of specific attributes for the query and repository components respectively, and $O_{ontology}$ is the time complexity for querying the COSMO ontology. However, $O_{ontology}$ cannot be calculated accurately because it depends on the reasoner employed to query the ontology, as well as the ontology structure in terms of the number or rules and objects in the ontology. In the current implementation we employ the Jena reasoner [59] in the calculation of MI to query the ontology about relations between attributes. Jena employs the Rete algorithm [41] for forward chaining to compile the ontology into rules, and backward chaining to derive new tuples from the rules, resulting in the worst case complexity being linear in the number of rules and polynomial in the number of objects, whereas in the best case the complexity is a constant [41]. However, it is difficult to estimate the value of $O_{ontology}$ because both the size (in the number of rules) and the content (in the number of objects) of the COSMO ontology grow with every addition of application domain and components respectively.

5.3.2 Experimental Analysis

COSMO Ontology for Queueing Networks

When a new application domain is added to the CoDES framework, the CoDES composition grammar and the COSMO ontology are extended with definitions for the new ap-

plication domain. The addition of a new composition grammar is discussed in Chapter 4. The addition of the queueing networks base components to the CODES repository is reflected in the COSMO Ontology as shown in Figure 5.4. *QNBaseComponent* is

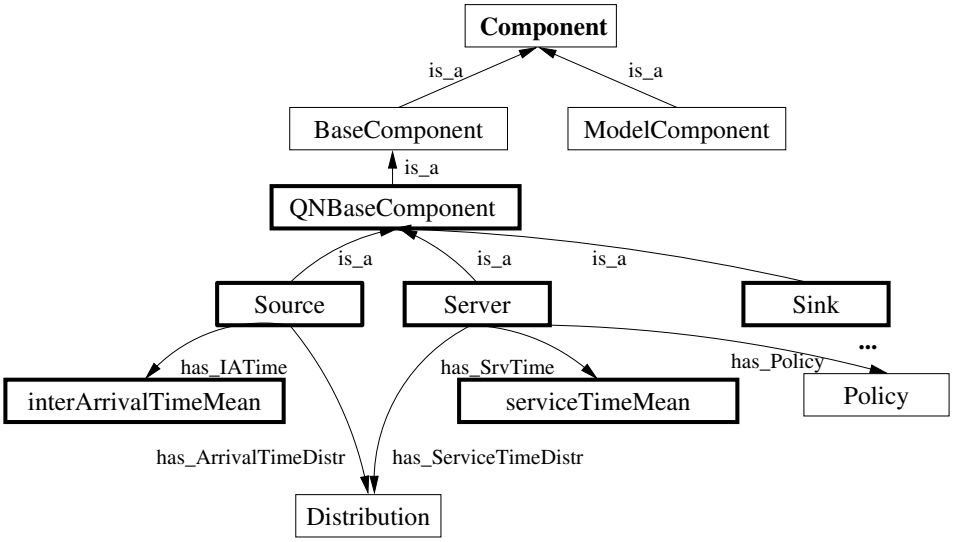


Figure 5.4: Extended COSMO Ontology for Queueing Networks

added as a new subclass of the BaseComponent 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 an 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.

Discovery using Matching Index

Consider model discovery for individual base components in the single-server queue example from Figure 4.3. Table 5.3 presents the query information provided by the user for component C_1, e.g. Source, compared to the component information for two

repository components. The first repository component, R_1, has the highest matching index and is returned by the discovery service. The second repository component R_2, has a lower matching index. The default weight values are $w_r = 0.33, w_a = 0.33$, and $w_b = 0.33$. However, weight values will be changed proportionally if one of the query parts is missing, such as specific attributes for C_1. Thus, we have for C_1:

$$MI(C_1, R_1) = 0.5 * 1 + 0.0 * 0 + 0.5 * 0.875 = 0.937.$$

An initial eliminatory search is performed, discarding base components whose type does not match the $Source$ type, which is derived from the conceptual model. $MI_r = 1$, since the type attribute matches exactly and all relevant words in the description of the query component are found in the repository component. Since no component attributes are provided, we have $MI_a = 0$. For the output constraints, the *destination* constraint matches exactly, and the *range = 21:24* of the repository component is included in the query range *range =* $10 : 35$, hence the term 0.75, with a total behavior matching index $MI_b = 0.875$.

	Query Component	**Repository Component**	**Matching Index**
	$\mathbf{C_1}$	$\mathbf{R_1}$	
Mandatory Attributes	**type:** Source **description:** open source	**type:** Source **description:** open source	$MI_r = 1$
Specific Attributes	-	-	$MI_a = 0$
Behavior: Input/Output Constraints	Input: $\emptyset$, Output: O_1 $IConstraints = \emptyset, OConstraints = \{O_{1C_1}\}$ $O_{1C_1} =$ { *destination = Server, range = 10:35* }	Input: $\emptyset$, Output: O_1 $IConstraints = \emptyset, OConstraints = \{O_{1R_1}\}$ $O_1R_1 =$ { *type = int, range =21:24, destination = Server*}	$MI_b = \frac{0+(1+0.75)}{0+2} = 0.875$
			$\mathbf{MI(C_1, R_1) = 0.937}$
	$\mathbf{C_1}$	$\mathbf{R_2}$	
Mandatory Attributes	**type:** Source **description:** open source	**type:** Source **description:** open source, two classes of jobs	$MI_r = 1$
Specific Attributes	-	-	$MI_a = 0$
Behavior: Input/Output Constraints	Input: $\emptyset$, Output: O_1 $IConstraints = \emptyset, OConstraints = \{O_{1C_1}\}$ $O_{1C_1} =$ { *destination = Server, range = 10:35* }	Input: $\emptyset$, Output: O_1, O_2 $IConstraints = \emptyset, OConstraints = \{O_{1R_2}, O_{2R_2}\}$ $O_{1R_2} =$ { *class = IO_Intensive, destination = Server*} $O_{2R_2} =$ { *class = CPU_Intensive, destination = Server*}	$MI_b = \frac{0+(1)}{0+2} = 0.5$
			$\mathbf{MI(C_1, R_2) = 0.75}$

Table 5.3: Query Information in a Single-Server Queue Model

In the second experiment, we evaluate the runtime cost for a randomly generated query request on a base component (query Q_1), and a query request on a model com-

ponent, (query Q_2), in the Queueing Networks application domain. For example, Q_1 could be a query for component C_1 as above, and Q_2 could be a query for the model component VO_1 in Figure 8.10(b). For each query, we measure the repository size, the number of repository components considered, the average time taken for the calculation of the Matching Index, and the time taken for the discovery service to return a candidate component. Table 5.4 shows an average of 10 runs for this experiment.

Query	**Repository Size**	**# Comparisons**	**MI Calculation Avg. Runtime** (s)	**Total Runtime** (s)
Q_1 (base component - *Source*)	1958	3	0.02	0.08
Q_2 (model component)	1958	1949	0.03	68.21

Table 5.4: Runtime Evaluation of Queries on Base and Model Components

As it can be seen, the average calculation of the Matching Index is similar for both base and model components. However, overall the discovery service fares much better in the case of base components than in the case of discovery of model components. This is a direct consequence of how the elimination strategy performs on this particular repository structure. In the case of query Q_1, the number of base components that have the same type as the query is significantly smaller (by a thousand factor) than in the case of query Q_2, which considers model components. Further improvements in the elimination strategy can be obtained by considering a more transparent abstraction of model components to include the model component inner structure. This implies a trade-off between a white-box and a black-box meta-component abstraction Other improvements can be obtained by storing the component repository in a relational database, rather than as a Java ArrayList as in the current implementation. This is a direction of future research and is discussed in detail in Chapter 9. Next, while the scale of the runtime cost has increased from milliseconds to seconds from syntactic composability verification to model discovery and selection, this cost is still feasible in the context of composed models with a reasonable number of components and a repository of medium size.

5.4 Summary

Key issues in model discovery and selection include the adequate representation of application domain and component knowledge, and a procedure to identify and rank partial matches. In order to meaningfully rank components in the repository based on a user query, we propose a Matching Index that uses semantically sugared attributes and behavior described in COSMO, our proposed component-based ontology.

COSMO is our proposed component-oriented ontology in which simulation components with *attributes* and *behavior* are classified as *base components*, specific to an application domain, and *model components*, which span multiple application domains. To the best of our knowledge, COSMO is the first component-oriented ontology specifically targeted towards component discovery and the meaningful validation of semantic composability. To meaningfully rank partial matches for a query component, we propose the *Matching Index* as a measure of component similarity. The Matching Index is derived using syntactic and semantic meta-component information, as a weighted sum of the three different component characteristics, namely mandatory attributes, specific component attributes, and component behavior. The similarity indexes are calculated based on relationships in the COSMO ontology.

Factors that influence the cost of the discovery and selection include the size of the component repository and the average number of specific attributes per component. The overhead incurred by the discovery service is small in the case of base components, but increases to up to one minute for model components. Simple heuristics have been introduced to reduce this cost, but improvements are still needed.

Chapter 6

Semantic Composability Validation

The previous chapters have seen the composed model evolve from an objective of the model composer, to a syntactically correct conceptual model in the CoDES framework, and lastly to a discovered model with attached implementation. Most component-based simulation and software engineering frameworks will stop here in the component-based life-cycle. However, the validation of the composed model is key to enhance the credibility of the composed model. In component-based modeling and simulation, the *semantic* composability of the composed model must be validated. As such, the validation process is called *semantic composability validation*.

In semantic composability, the composition must be meaningful for all components involved. Furthermore, the composed model must be valid [88]. This is because simulation models are widely used to make critical decisions and to answer "what-if" questions [11]. In modeling and simulation in general, a valid simulation model is one that mimics closely the real system that the simulation model abstracts [10]. Here, while overall program correctness is required, it is very important for the simulation to produce results that are close to those obtained in the real system it models. Very often, this similarity cannot be fully captured by an automated validation process because it refers both to input/output transformations, i. e. the simulation model must have the

same output as the real system when presented with the same input, as well as finer points such as overall simulation model state and unified component assumptions and context [30, 126]. Furthermore, a system expert is required when the simulation model is used in critical situations where a valid answer is crucial, such as in military training simulations [51, 72]. For example, the process of Verification, Validation and Accreditation (VV&A) for modeling and simulation in the US Department of the Navy, which must be performed for all models and simulations [130], defines seven user roles and thirteen important steps grouped in five categories, namely planning, conceptual model validation, design verification, implementation verification, and results validation [35]. As such, the main design considerations and trade-offs in semantic composability validation become accuracy and cost.

The semantic validation of composable simulations is a non-trivial problem [11, 30, 88, 126]. Challenges arise from the fact that composition is not a closed operation with respect to validation because valid components do not necessary form valid compositions [10]. This means that the validation process must look at the overall semantic behavior of the entire composed model, and not at the individual semantic validity of its composing components. Next, reused components are developed for different purposes and when composed may result in emergent properties [49]. This implies that the overall behavior of the composed model cannot be obtained as a union of the individual behaviors of its constituents, because the interaction of the components over time results in properties that are not evident in the individual components. Similarly, the context in which a reused component was developed and validated might differ from the new context of the composed model [12, 126]. This means that the new context in which the components are executing can influence their interaction in unspecified ways. Next, there exist various validation perspectives on the component interactions over time. The validation process must address *model behavior* aspects such as deadlock, safety, and

liveness, *temporal* aspects such as the behavior of components and compositions over time, and *formal* aspects such as the need to provide a formal measure of the validity of compositions, also called "figure of merit" [61].

The motivation of our work is twofold. Firstly, simulation model validation is a lengthy, manual process that can be improved if an automated process that focuses on the behaviors of individual components as well as of the composed model is applied. Secondly, well-established software verification techniques, such as model checking, can be adapted to the simulation validation perspective to increase the credibility of the validation process. In composable simulations, the main validation techniques include formal methods such as the DEVS formalism [132], Petty and Weisel's theory of composability [88], and component abstractions such as BOM [75].

This chapter starts with an overview of current approaches to the validation of semantic composability of the composed model. We continue with an overview of our proposed strategy and associated proposed techniques for the validation of semantic composability. For ease of reading, we present the validation of general model properties, which makes up the first layer of our validation process, in Section 6.2. The second layer is presented in detail in Chapter 7.

6.1 Current Approaches

Petty and Weisel pioneered a formal theory of composability validation, which allows for a composed simulation model to be checked for semantic validity [88]. They provide definitions for components, simulations, and validity, and establish the basis of a theory of semantic composability validation. At the heart of the theory lies a component formalism in which a simulation component is represented as a mathematical function. Next, composition is modeled as a mathematical functional composition. The simulation of a composition is represented as an LTS where nodes are model states, edges are

function executions, and labels are model inputs. A composition is valid if and only if its simulation is close by a relation to the simulation of a perfect model. The fundamental drawback of this approach is that time is not modeled and the function representing a component makes an instantaneous transition from input to output. This permits only a static representation of the composition. Furthermore, the LTS representation considers the functions strictly in the order they appear in the mathematical composition, which might not be accurate for complex compositions. For example, an integer functional representation and a composition formalism that lacks time does not fare well in compositions that have "fork" and "join" connectors because the timely passage from one branch to another branch of the connectors cannot be specified.

Another approach to the validation of semantic composability is to look at the definition of the term "valid" from a strictly software engineering perspective. In software engineering, a valid program is one in which the process interaction follows some protocols or specifications [63]. As such, formal theorem proving tools or model checkers can be used to verify desired properties of the composed model. For example, in recent work that proposes to formally validate compositions of DEVS models, formally represents the DEVS model in the Z specification language [127]. A theorem proving tool based on Z such as Z/EVES [100] is used to verify the model and discover hidden properties. Ambiguities, conflicts and inconsistencies can be discovered in the specification. However, the Z specification language lacks time modeling, a most important attribute in DEVS models. As such, the validation process is incomplete.

A third approach to composition validation [75] uses the Base Object Model (BOM) [50] as a component abstraction. A key assumption in this approach is that a valid composition is represented using a detailed user specified composition scenario. The scenario includes the sequence of component execution, as well as events and parameter names for interacting components. Component discovery is performed based on the

specified scenario. A valid composition of discovered components is one in which the sequence of actions or events is the same as or includes the sequence specified in the scenario. However, the somewhat informal validation process includes the composition and execution of discovered components in *all* possible combinations in order to be compared with the specified scenario. Furthermore, a detailed execution scenario might not be available from the model composer.

6.2 Validation Strategy Overview

We start by providing a definition of semantic composability validity.

Definition 10 (Semantic Composability). We consider a composition to be **valid** and its components to be **semantically composable** if and only if (i) components to be integrated *behave* correctly to form a valid composition both *externally* with respect to their neighbors, and *internally* when safety and liveness properties are preserved over time, and (ii) the resulting composition produces valid output.

This definition considers the validity of the composed model as a whole, as well as the valid behavior of the individual components while they are interacting with their neighbors across the simulation time.

As discussed above, the semantic validation of composed models is a compute-intensive process [10, 30, 88]. This is because of many factors, which include among many the various validation perspectives, the size of the composition in terms of attributes and behavior, and the difficulty in obtaining a resultant behavior for the composition based on the behavior of its components. Next, for a simulation problem, the population of models that abstract it often includes more invalid than valid models. These models are invalid either because of incompatibility between communicating components, or interleaved execution issues such as deadlock or safety and liveness

issues. Furthermore, the question of validity does not have a simple, clear-cut, yes/no answer. On the other hand, formal guarantees of the validity of the composed model are required to increase the credibility of the model.

As such, we propose a two-layered *deny validity* validation strategy with increasing accuracy and complexity and incremental cost, as shown in Figure 6.1. Given that the model population for a simulation problem consists of both valid and invalid models, our key strategy is to incrementally discard invalid models that do not meet several properties. Firstly, syntactically incorrect models are discarded by syntactic composability

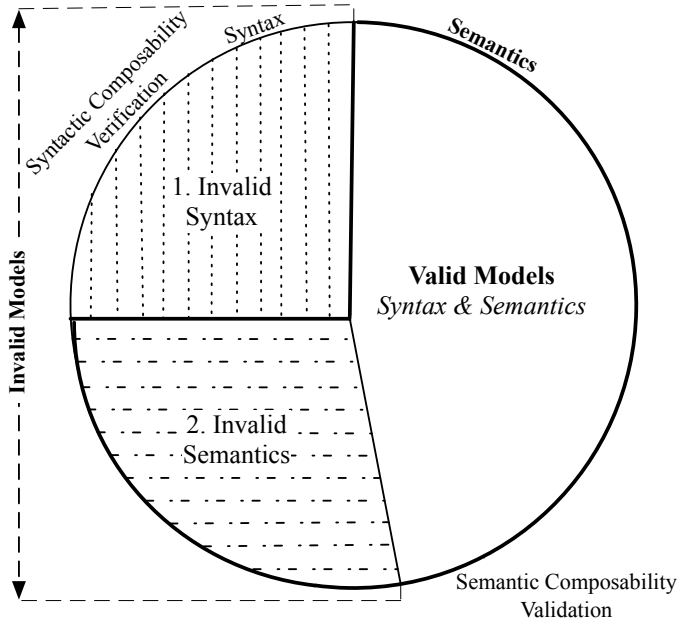


Figure 6.1: Composable Model Population

verification in the Model Composition and Verification stage, as discussed in Chapter 4. What is left is a population of models that has valid models (both syntax and semantics), and models with valid syntax but invalid semantics. Our approach aims to remove the models with valid syntax but invalid semantics in two steps. Informally, models with invalid semantics include models in which components cannot understand each other, or models in which components are able to communicate meaningfully, but for which the interleaved execution of the components over time leads to deadlock or to invalid safety

and liveness properties. As such, we include in our validation process a layer that aims to discard models for which these general properties are not met. Another type of models with invalid semantic composability are those that have valid model properties, but whose execution is not close to that of the real system the composed model abstracts. To eliminate such models, we perform formal validation with respect to a reference perfect model. In contrast to current static perfect model validation, our novel *time-based formalism* represents dynamic component behavior. Furthermore, we are able to quantify the similarity between the composed model execution and the reference model execution using our defined semantic metric relation.

Our proposed strategy translates into a two-layered validation process as shown in Figure 6.2. We first validate general model properties, which include semantically correct component communication, as well as safety and liveness of the interleaved execution of the components over time. In the first step, *Validation of Component Communication*, we discard models for which the component communication is not compatible [125]. Next, we validate the composed model for general properties including safety and liveness for instantaneous transitions and over time. *Concurrent Process Validation* validates that the component communication is correctly coordinated, regardless of time considerations or specific computations that the components might perform. If this is true, we introduce the concept of time in *Meta-Simulation Validation* (MSV), and validate safety, liveness and deadlock freedom using sampled time values for the time attributes [119]. In *Formal Validation of Model Execution*, we evaluate if the execution of the composed model is exactly the same to that of a reference model [122]. If this is not true, we try to establish if the execution of the composed model is close enough to that of the reference model.

We continue by presenting the first layer of our approach, namely the validation of general model properties. The formal validation of model execution is presented in

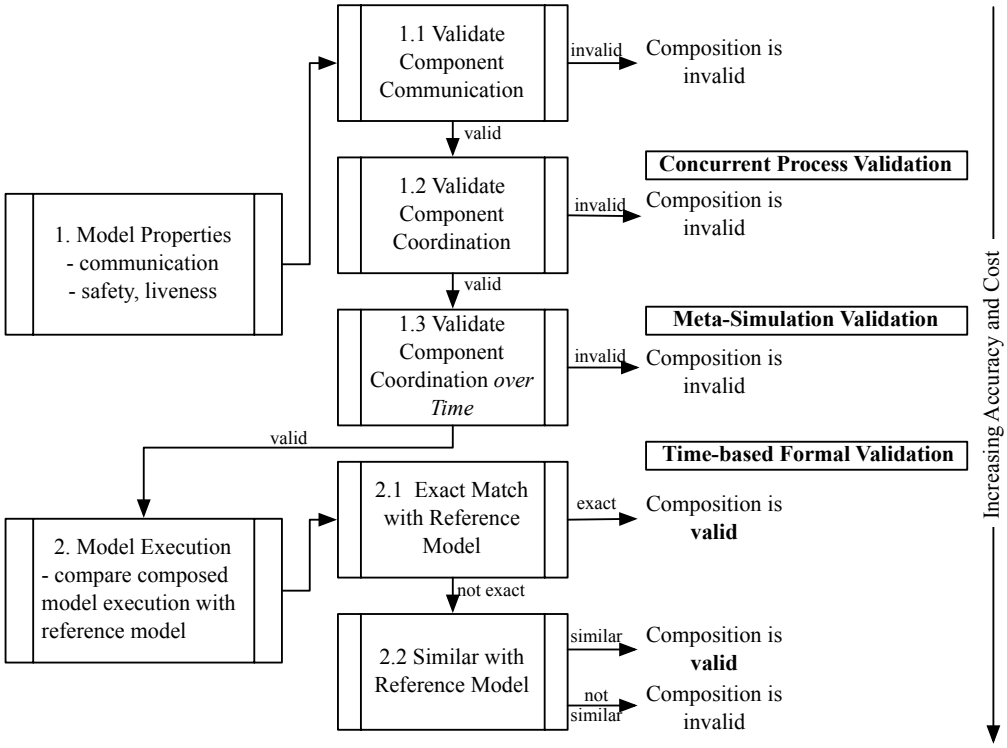


Figure 6.2: A Layered Deny Validity Approach to Semantic Validation

Chapter 7.

6.3 Model Properties

6.3.1 Component Communication

The first and simplest class of models with invalid properties is that in which the components cannot communicate properly. This can happen despite the components exchanging messages using the same syntax (validated by syntactic composability verification). We validate therefore semantic compatibility between any two neighboring components is correct, i.e., that the data that the component on the sending end sends is consistent with the data that the component on the receiving end expects. The notion of "consistent" here refers to consistency with respect to data types defined in the COSMO ontology. It is important to highlight here that in this first validation step we are not yet

considering time. As such it might be the case that a semantically correct communication, validated in this first step, becomes invalid when the timed arrival or departure of messages is considered.

A subtle difference between this step and syntactic composability verification is worth highlighting. In syntactic composability, by ensuring that components are represented and connected using the standard COML format, we ensure that components communicate using the same syntax. In contrast, in this step we validate whether the component communication is meaningful. In other words, syntactic composability verification ensured that components speak the same language, whereas the validation of component communication ensures that components understand each other.

We propose a *Composability Index* (CI) to measure semantic compatibility with respect to communicating components. As shown in the pseudo-code from Figure 6.3, the composability index of the composition is calculated using local composability indices for each pair of neighboring components. As shown in line 6, for each connected

```
1.  double calculateLocalCI(Component a, Component b){
2.          return CI(a.output, b.input, COSMO); }
3.
4.  double constraintValidation(Composition model){
5.          foreach(ConnectedComponentPair pair){
6.                  ci += calculateLocalCI(pair.a, pair.b);
7.          }
8.          return ci/sumConstraints;
9.  }
```

Figure 6.3: Validation of Component Communication

pairs of components, we calculate a semantic composability index, which is a measure of how well the components can communicate. A global composability index is calculated as the sum of the local composability indexes over the total number of constraints, as explained in Definition 11. If $CI = 1$, then the semantic composability validation process can continue as shown in Figure 6.2.

Definition 11 (Composability Index). Let $Comp = \{(C_i, C_j) | C_i = \langle R_{C_i}, A_{C_i}, B_{C_i} \rangle, C_j =$

$\langle R_{C_j}, A_{C_j}, B_{C_j} \rangle\}$ be the composition of connected components (C_i, C_j), with the output of C_i transformed into the input of C_j. Let OC_{C_i} be the constraint set on the output data for component C_i, and IC_{C_j} be the constraint set on the input data for component C_j. The composability index CI of the composition $Comp$ is defined as

$$CI(Comp) = \frac{\sum_{ic \in IC_{C_j}} SAT(ic, OC_{C_i})}{\sum_{(C_i, C_j) \in Comp} |IC_{C_j}|},$$

where $(C_i, C_j) \in Comp$, and

$$SAT(ic, OC_{C_i}) = \begin{cases} 1 & \exists\, oc \in OC_{C_i} \text{ such that type(oc) = type(ic),} \\ & \text{and oc satisfies ic} \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

For each pair of connected components (C_i, C_j), we measure the number of constraints on the input of C_j that are satisfied by the output of C_i.

The composability index is calculated as a fraction of the satisfied constraints of all connected components from all the constraints between connected components. However, the constraints that do not have a corresponding constraint type in the neighboring components cannot be verified but might contribute to the semantic validity of the composition. For now unsatisfiable constraints are ignored but the simulator developer can be involved in establishing their relevancy. Additionally, the composability index formula measures the number of satisfied constraints on input/output data for all connected component pairs, but more complex formulae can be devised.

The component with the highest MI for a given query is automatically selected in individual component discovery. However, this may not result in a composition with $CI = 1$. If a $CI \neq 1$, candidates with lower MI are selected and the composition

is verified again for data compatibility. This facilitates the evalation of the suitability of different compositions for the same query. Moreover, all validation steps to follow will assume that the component communication is semantically correct. As we will see in Section 6.4.1, this assumption helps to reduce the time complexity of the following validation steps.

6.3.2 Concurrent Process Validation

In *Concurrent Process Validation*, the logical correctness of component coordination is validated. This layer guarantees that *safety* and *liveness* properties hold for any possible interleaved execution of the concurrent processes. Furthermore, we check that the composed model is deadlock free in the context of *instantaneous* transitions. By *safety* we mean that the component does not invalidate some logical properties, and *liveness* guarantees the component reaches a specific pre-defined state in *all* execution traces. However, in application domains such as open queueing networks, deadlock is not inherent and as such need not be validated. We employ model checking techniques to perform our desired validation. A composed model is invalid if it is found to be deadlocked, or if any of the components invalidate their safety or liveness properties. Model checking is employed because it is timeless, automatically performs deadlock verification, and considers all interleaved execution traces.

As shown in Figure 6.4, the behavior of each meta-component modeled as a state machine is translated into a logical specification using a logic converter module. Different converters can be developed for each application domain and targeting various logical properties. The converter takes as inputs the meta-components and the composition topology. The result is a specification describing the composition together with an expression of the safety and liveness properties. To prevent state explosion, each component's state machine is reduced by considering only communication states and

attributes that influence state transitions. We discuss state explosion in model checking in Chapter 8. The actions of non-communicating states are abstracted as a single atomic operation. Similarly, time is not modeled and transitions are considered instantaneous. This layer assumes that the communication between states is meaningful and correct. This allows for the abstraction of data sent through communication channels to a single type of message[1].

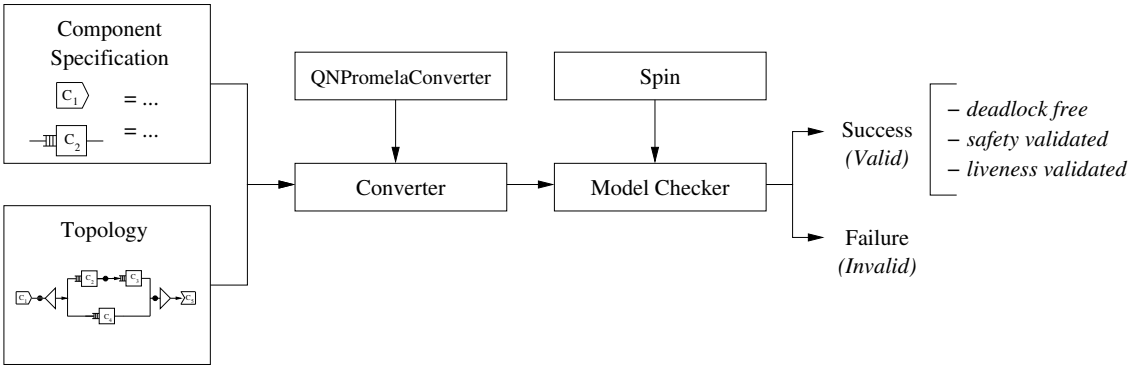


Figure 6.4: Concurrent Process Validation

The resulting specification is then verified using a model checker. For example, a Promela Converter can be used to translate the topology into a Promela specification [53], and the Promela specification can be validated using the SPIN model checker [14]. However, other model checkers such as UPPAAL [65] can be employed. In model checking, the entire state space containing all possible interleaved executions of all concurrent processes is analyzed exhaustively for deadlock, safety, and liveness. For example, in the Promela language we specify various safety properties using simple *assert* statements. Liveness properties (e.g. a component outputs at least one message) are specified using *progress* labels. In the validation process, if the *assert* statement is not verified in any of the states, the model checker issues a safety error. Furthermore, if there exists a cycle that does not visit the *progress* label infinitely often, then a liveness error is issued. An important observation is that discrete time modeling is possible in

[1]This has the advantage of reducing state explosion.

some specification languages [16]. Unfortunately, these solutions are somewhat obsolete and work only for small systems and toy examples. A simpler way to model time is by attaching an additional counter process to each main process. However, our results show that the state space explodes even for systems with as few as three components. As such, we integrate counter processes only for components that inject jobs into the system, i.e., for components with only output channels. Additionally, symbolic execution can be used to further reduce the state space, while at the same time enhance expressivity [107]. We leave symbolic execution of composed simulation models as part of our future work.

The model checker validates that safety, liveness, and deadlock-free properties hold for the abstracted composition. Constraints such as non-floating point types and unbounded queues limit the computational expressivity of the specification but do not hinder the validation of the component coordination. However, state explosion is an important issue. We limit state explosion by considering only important attributes and a limited number of counter processes. Furthermore, artificial termination conditions, such as allowing components to process a limited number of messages, can be added.

6.3.3 Meta-Simulation Validation

Concurrent process validation establishes that all instantaneous interleaved traces of the composition execution are correct. However, it does not consider time and other component attributes that might influence the composition run. This stage validates that the logical properties demonstrated in the previous layer hold throughout the simulation run. State machines of meta-components together with all time delay mechanisms and other participating attributes ignored in the previous layer are executed concurrently in a *meta-simulation* to validate properties such as safety and liveness. Time attributes that model time delaying mechanisms (e.g. inter-arrival time, service time) are sam-

pled, if necessary, from specific distributions detailed in the meta-component. Since this method is inherently based on sampling, more than one meta-simulation run is performed with various time attribute values.

Safety properties are specified through *validity points* provided by the user. A validity point is a connection point in the topology through which a certain type of data must pass. Safety errors are issued if incompatible data flows through the validity points at any point during the meta-simulation run. Liveness is validated by assigning a *transient* predicate to each component. A component specific transient predicate guarantees that if it becomes true during the meta-simulation, then it will become false before a *timeout* elapses. The transient predicate is defined such that a change in its truth value signifies a change in the component state or attributes. The value of the transient predicate is ideally given by the component creator in the meta-component, but it can also be deduced from the state machine. The timeout interval can be characteristic to the composition or specific to each component.

The state machine specified in the meta-component, including attributes and transitions ignored in the previous layer, are translated into a class hierarchy and subsequently run. *Time attributes* that model time delaying mechanisms (e.g. inter-arrival time, service time) are also considered. All time attributes are sampled, if necessary, from specific distributions. The distribution type, as well as its mean value, are also attribute values in the meta-component. Since this method is inherently based on sampling, more than one meta-simulation run is performed with various time attribute values. If any of the meta-simulation returns an error, then the composition is considered invalid.

Figure 6.5 presents the pseudo-code for the Meta-Simulation validation process implemented in Java. In the `init` stage, each COML component is transformed into a Java Component class as shown on line 4. Its attributes as well as its behavior modeled as a state machine are also set. Next, a liveness observer is created using the compo-

```
1  init (COMLComponentList compList, ConnectorList connList){
2        foreach (COMLComponent comp in compList){
3              //create new component class
4              c = new Component(comp); //create attributes and behavior
5              c. setTransient (comp.getTransient ());
6              //add liveness observer
7              c.addLivenessObserver(new LivenessObserver(c. getTransient ()));
8              //add the component to the component list
9              componentList.add(c);}
10       foreach (Connector conn in connList){
11             //add & link components on the right and left side
12             addRightComponents(conn.getRightComps(componentList));
13             addLeftComponents(conn.getLeftComps(componentList));
14             //add validity point observer
15             foreach ( ValidityPoint vp in conn.getVP()){
16                   addVPObserver(new VPObserver(vp));}}
17 }

19 run(ComponentList compList, ConnectorList connList){
20       foreach(Component comp in compList)
21             start thread (comp)
22       foreach(Connector conn in connList)
23             start thread (comp)
24 }
```

Figure 6.5: Meta-Simulation Pseudo-code

nent's transient predicate on line 7. Validity points observers are added if the connector has any validity points set. In the `run` stage, the components and connectors are started on individual threads and their assigned observers collect relevant safety and liveness information.

6.4 Theoretical and Experimental Analysis

6.4.1 Theoretical Analysis

We analyze the time complexities of Concurrent Process Validation (O_{CP}) and Meta-Simulation Validation (O_{MS}) validation. Let n be the total number of components and c the total number of connector branches in the composed model.

Time Complexity of Concurrent Process Validation

In Concurrent Process Validation, the entire state space of the composition is exhaustively verified. Thus, the time complexity directly depends on the size of the state space, $O_{CP} = O(\mathbb{S})$, where $\mathbb{S}$ is the size of the state space. The state space can be divided into two main parts, the components and communication channels that link them. We consider the communication channels separately because they are important entities on which the component synchronization is done. Thus we have

$$\mathbb{S} = \mathbb{S}_{comp} \times \mathbb{S}_{chan}$$

where $\mathbb{S}_{comp}$ is the size of the state space of the components, and $\mathbb{S}_{chan}$ is the size of the state space of the communication channels. Considering that each component i has T_i different simulation states (where *simulation state* means the collection of all component attributes and their values at a particular observation moment), the value for $\mathbb{S}_{comp}$ is:

$$\mathbb{S}_{comp} = \prod_{i=1}^{n} T_i$$

In the calculation of $\mathbb{S}_{chan}$ we consider that the q communication channels are bounded, allowing a maximum number s of messages, and that in each communication channel there can be m different types of messages. Thus,

$$\mathbb{S}_{chan} = \prod_{i=1}^{q} \sum_{j=0}^{s} m^j = (\sum_{i=0}^{s} m^i)^q$$

Therefore, the size of the state space becomes:

$$\mathbb{S} = \prod_{i=1}^{n} T_i \times (\sum_{i=0}^{s} m^i)^q$$

The component communication is already validated by the Constraint Validation pro-

cess. Therefore, irrespective of the specification language and model checker used, m can be reduced to a single type of message, i.e., $m = 1$. Considering the worst case where $q = 2 * c$, and the trivial observation that the number of connectors is a polynomial of grade 1 of the number of components, $c = P^1(n)$, then

$$O_{CP} = O(s^{2P^1(n)} \times \prod_{i=1}^{n} T_i) \tag{6.2}$$

The right-hand side product depends on the type of abstractions employed when the COML component specification is translated into the model checker specification. Regardless, Concurrent Process Validation is *exponential* in the number of components in the composition.

Time Complexity of Meta-Simulation Validation

Intuitively, in the average case the complexity of the Meta-Simulation layer is by design much smaller than O_{CP}. This is because the state space is not exhaustively parsed since the simulation is run for a limited amount of time and transitions between states are not instantaneous. Furthermore, when considering the liveness of a component, the Meta-Simulation approach considers only the two possible truth values of the *transient* predicate. As such, we have:

$$\mathbb{S}_{comp} = \prod_{i=1}^{n} 2 = 2^n$$

Next, when we validate that components do not stall during their run, the different types of messages that could be in the communication channels are not important. The only significant information about the communication channels is whether they are empty. Therefore,

$$\mathbb{S}_{chan} = 2^c$$

Thus the complexity of Meta-Simulation Validation becomes:

$$O_{MS} = O(2^{n \times c}) = O(2^{n \times P^1(n)}) \tag{6.3}$$

6.4.2 Experimental Analysis

We demonstrate our approach using a simple single-server queue example presented in Figure 4.7. Key meta-component information for each component is shown in Table 6.1. To focus on our approach, we consider only simplified component state machines.

	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$
Attribute	$noJobsGenerated = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ interArrivalTime: exponential(3) $transient(C_1) : (noJobsGenerated == 1)$	$noJobsServiced = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ serviceTime : exponential(6) $busy = false$ $transient(C_2) : (busy == true)$	$noJobsPrinted = 0$ $timeout = 20$ $time = 200$ $timeScale = 1/5$ $\Delta printingTime = 1$ $transient(C_3) : (noJobsPrinted == 1)$
Input	-	I_1, constraints: $origin = Source\|Server$ $range = 10; 35$ $type = double$	I_1, constraints: $origin = Server$
Output	O_1, constraints: $destination = Server$ $range = 11; 15$ $type = int$	O_1, constraints: $destination = Server\|Sink$ $range = 10; 20$ $type = double$	-
State Machine	$S_1(\Delta interArrivalTime) \rightarrow S_2$ $S_2 \rightarrow S_1O_1[A_1]$ $[A_1] = noJobsGenerated++;$	$I_1S_1 \rightarrow S_2[A_1; A_3]$ $S_2(\Delta serviceTime) \rightarrow S_1O_1[A_2]$ $[A_1] = (busy = true);$ $[A_2] = (busy = false);$ $[A_3] = noJobsServiced++;$	$I_1S_1 \rightarrow S_2$ $S_2(\Delta printingTime) \rightarrow S_1[A_1]$ $[A_1] = noJobsPrinted++;$

Table 6.1: Meta-component Information

Component Communication

To validate the semantic correctness of component communication, we calculate the composability index CI for the composed model in Figure 4.7 according to Definition 11 as follows:

$$CI(Comp) = \frac{SAT(ic_2, OC_{C1}) + SAT(ic_3, OC_{C2})}{2} = \frac{1+1}{2} = 1$$

The component communication is validated and as such the validation process can continue with the validation of component coordination in concurrent process validation.

Concurrent Process Validation

In Concurrent Process Validation, a Promela Converter can be used to translate the topology into a Promela specification, and the Promela specification can be validated using the SPIN model checker [14]. Figure 6.6 shows the state machine of every component in the single-server queue example translated into a Promela specification.

```
1  mtype {Job}; chan to1 = [10] of {mtype}; chan to2 = [10] of {mtype}; ...
2  hidden byte sourceIAMax = 10; byte sourceIATime; byte noJobsSource = 0;
3  proctype CON_ONE_TO_ONE(chan in, out){
4      do :: in ? Job -> out ! Job; od}
5
6  proctype SOURCE(int id, noJobsMax; chan out){
7      do :: (sourceIATime == sourceIAMax) -> sourceIATime =0;
8      if :: out ! Job -> progress: printf("[Source] Job sent"); fi od } ...
9  active proctype monitor(){assert (noJobsSource < noJobsMax);}
10
11 proctype SINK(){...}
12
13 proctype SERVER(int id; chan in, out){
14     S1: {if :: in ? Job -> printf("[Server] Job received!"); busy=1;goto S2; fi}
15     S2: {if :: out ! Job -> progress: printf("[Server] Job sent! "); busy=0; goto S1;}}
16
17 init{
18 run SourceCounter(); ...
19 run SOURCE(1, from1);
20 run CON_ONE_TO_ONE(from1, to2);
21 run CON_ONE_TO_ONE(from2, to3);}
```

Figure 6.6: Single-Server Queue Model in Promela

Each state is transformed into a Promela label, which includes conditions and input and/or output actions as specified by the meta-component behavior. Transitions between states are assumed to be instantaneous. Nonetheless, for component C_1 described in process *SOURCE1* on line 6 we simulate time through the additional process

SourceCounter shown on line 9. Counter processes are introduced for all components that have only output channels. Each type of connector is defined as a Promela process. For example, process *CON_ONE_TO_ONE* on line 3 describes the one-to-one connector. The fork and join connectors are not part of this composition and as such are omitted. In the `init` method on line 18, communication channels are assigned to the connectors and components according to their connection topology. Similar to the behavior of connectors in the real system, communication in our Promela specification is asynchronous. However, the maximum number of messages in a channel is bounded by a constant value. This is because unbounded queues are not permitted in the SPIN model checker [14], since the focus is on process *coordination* and not computation.

Valid executions can be specified in Promela and subsequently validated by the SPIN model checker. To specify safety, we create `assert` statements such as the one on line 9 for important properties. To specify liveness, we assign a `progress` label to each state in a component that produces output, such as the one on line 8. By default the SPIN model checker validates that there is no deadlock or any unreachable states in the system. The SPIN model checker validates the system by analyzing all possible process states obtained through the interleaved execution of the active processes. State space explosion decreases the feasibility of employing this type of validation as a standalone validation process, and thus we include it only as the first layer in our approach.

Meta-Simulation Validation

Meta-simulation validation shows that the logical properties demonstrated in the previous step hold through time. Our implementation translates the complete state machine of each component into a Java class hierarchy. Attributes and their values provided by the user, state transitions, as well as time are modeled. Next, we construct a meta-simulation of the composed model using the translated classes. We execute the meta-simulation over the simulation time and observe the execution for desired properties.

During the meta-simulation run, sampling is performed for attributes that require so. This is the case especially for time attributes such as inter-arrival time or service time. For example, as shown in Table 6.1, the inter-arrival time $\Delta interArrivalTime$ for component C_1 is sampled from an exponential distribution with a $mean$ of 3. The distribution type and mean values are an example of attribute values provided by the user. Since sampling is performed, the meta-simulation is run for $N = n * noSampling$ times, where n is the total number of components and $noSampling$ is the total number of locations where sampling is done. If any of the properties does not hold in the meta-simulation runs, the composition is declared invalid.

The most important logical properties that are validated through time are safety and liveness. From a practical perspective, we consider safety to mean that components do not produce invalid output. The simulator developer specifies the desired valid output by providing *validity points* at various connection points in the composition. A validity point contains semantic description of data that must pass through its assigned connection point. For example, the two validity points for the data that passes through the second connector in Figure 4.7 could be $VP_1 = d_1\{origin = Server, destination = Sink, range = 10; 35, type = double\}$, and $VP_2 = d_2\{origin = Server, destination = Sink, range = 1; 2\}$. If anytime during the meta-simulation run, semantically incompatible data passes through the connection point, a safety error is issued. Semantically incompatible data is data whose type and constraints are not related in the COSMO ontology.

Liveness is validated by considering a *transient* predicate assigned to each component. The value of the transient predicate is ideally provided by the component creator in the meta-component as shown in Table 6.1. Its initial value is `false`. Each component is assigned a *liveness observer* that is notified every time the attributes involved in a transition change values. The liveness observer evaluates the transient predicate and

time stamps the moment in which the transient predicate becomes true. A component is considered *alive* if its liveness observer has evaluated the transient predicate to *true* and then to *false* in an interval of time smaller than the specified timeout. For example the transient predicate for component C_2 could be $transient(C_2) = (busy == true)$.

Based on the meta-component information from Table 6.1, the state machine for each component is executed on separate threads. Figure B.2 presents one of the meta-simulation runs for our example model. The flow of input and output from component to connector and reverse ([$Connector$]) as well as the execution of the safety and liveness ([$Observer$]) observer are shown.

Generating Semantically Valid Models

In Section 4.3.2, we were able to obtain a repository of syntactically correct, standalone simulators, by automatically generating them from the Queueing Network Composition grammar using queueing networks base components.

To create semantically valid models with respect to model properties, we assign each base component with a *COML* file selected from a pool of sample COML files for that type of base component. We perform this assignment repeatedly until the composed model passes the validation of general model properties. For Meta-simulation validation, in which attribute values are needed to perform sampling, we employ the default values of component attributes, and consider only meta-components that have such values provided.

Specifically, the component communication is semantically validated. If the simulator has a $CI = 1$, the process is followed by Concurrent Process Validation and Meta-Simulation validation. Only simulators that pass all the tests are saved in the repository. In order to obtain model components, several components from the validated simulators must be stripped in order to obtain model components with both "in" and "out" communication channels. In the case of the simple composition grammar

employed in Section 4.3.2, in which open queueing models with only one Source and one Sink are generated, the stripping of components from the simulator to create model components is effortless, since only the Source and the Sink components are removed.

Table 6.2 presents the number of semantically valid model components with the total number of components ranging from 4 to 10. The total number of generated model

#Components	**#Models with Correct Syntax**	**# Models with Valid Semantics**
4	2	2
5	6	6
6	19	19
7	57	57
8	164	164
9	457	457
10	1,244	1,244
Total	1,949	1,949

Table 6.2: Number of Model Components in the Repository

components adds up to 1,949 model components, for a total of 1,958 components in the CoDES repository for the Queueing Networks application domain. In our sample case, we were able to obtain the same number of semantically valid models as the syntactically correct models.

6.5 Summary

This chapter presents our approach for the validation of semantic composability. For a composed and discovered model, we attempt to discard it as invalid using a new dual-step *deny validity* process. Firstly, we attempt to discard models that have invalid general model properties, such as component communication, and logical properties such as safety and liveness of instantaneous and timed transitions. If the composed model passes this test, we next formally compare the execution of the composed model over time with that of a reference model deduced according to the composition structure.

This second layer is presented in detail in the next chapter.

In the validation of general model properties, we target several key properties. Firstly, we validate that the component communication is semantically meaningful. We propose the *Composability Index*, a new measure of the semantic correctness of component communication. The composability index is calculated as a fraction of the satisfied constraints of all connected components from all the constraints on input/output data exchanged between connected components. These constraints are semantically defined in our proposed COSMO ontology.

Next, we validate the *logical* coordination of components in the composed model. We focus on properties such as safety and liveness in the context of instantaneous and timed transitions. In *concurrent process validation*, we employ model checking to validate that a composed model satisfies safety, liveness and deadlock-freedom using abstracted component representations that focus only on component coordination, by removing timed transitions and other component attributes. If a model passes the concurrent process validation stage, the *meta-simulation* validation step introduces timed transitions and other component attributes to validate the model over time, using definitions of safety and liveness as validity points and transient predicates respectively.

Chapter 7

Formal Validation of Semantic Composability

In the previous chapter, we proposed a deny validity approach for the validation of semantic composability. Given a composed model, we try to discard it as invalid using a series of validation tests. We base our approach on the common sense observation that there are various degrees of semantic validity and, more importantly, that there are more invalid models than valid models for a simulation problem. Firstly, we try to validate general composition properties, such as the semantic correctness of component communication, and the valid component coordination in the context of instantaneous and timed transitions. If the composed model fails any of these tests, it is discarded as invalid.

However, it is possible for a model to be invalid even if it has the desired model properties mentioned. This is the case where the composed model does produce results, but these results are not close to those produced by the real system that the composed model abstracts. In traditional modeling and simulation that does not adhere to a component-based paradigm, this problem is addressed through the validation of input/output transformations [11]. However, the comparison of input/output transforma-

tions is not a sufficient measure of composition validity, especially since the combined output of the entire composition is not a direct reunion of the outputs of the individual components. We thus propose to compare the execution of the composed model with the execution of a reference model over time, using a formal validation method.

The main purpose of formal model execution validation is to obtain a guarantee of the composition validation that relies on a formal theory, by analyzing the similarity between the composition execution and the execution of a reference model. Furthermore, we propose to obtain a quantitative measure of this similarity. Towards this, we propose a formalism to describe composability and validity. This formalism contains definitions of simulation components, simulation, and validity to facilitate reasoning about the composition validity. Current approaches that attempt to formalize semantic composability validation [88] fail to capture important characteristics of simulation components such as time and state in their formalism. This high level of abstraction facilitates the understanding of the formalism and reasoning about complex properties such as different types of validity under composition, but is not applicable to real simulation scenarios. In contrast, we represent a component as a function of states over time and formalize composition, simulation and validity accordingly.

Our five-step formal validation process relies on our proposed formalism. The proposed validation process aims to provide a formal measure of composition validity by comparing the composed model with a reference model. One of the problems that arise here is the source of the reference model. One solution would be to require the model composer to provide the reference model. However, this process is tedious and error-prone. As such, we propose to obtain the reference model from reference components in the repository. We consider that for each type of base component, there exists a reference model in the repository, initially provided by domain experts when the application domain is added to the framework. The reference base component models describe

what the domain experts consider to be the ideal component behavior. The generic descriptions lack specific attributes (e.g. sampling distributions for time attributes) and *are without an implementation*. We assume that for each base component type (e.g. *Source* in Queueing Networks Application domain), there exist different base component implementations in the repository (e.g. *SourceOpen* - a *Source* component for open queueing network systems).

A reference component is a generic, desirable representation of a base component ideally provided by domain experts when the new application domain is added to the framework. Ideally, the reference components should describe what the system experts consider to be the desirable base component behavior. It should be generic in the sense that their description lacks any real data values. It follows that the reference model composed from the generic reference components is only a description of the desired simulation, without an attached implementation. Throughout the validation process, the generic reference components attributes will be instantiated using the same attribute values used by the corresponding components in the composed model. Lastly, the base component implementations may differ widely from the reference base component models.

The idea of comparing a composed model with a reference model has been previously explored in [88], which represent components as functions of integer values. This approach is presented in detail in Chapter 6. However, this representation does not allow complex compositions (e.g. with "fork" connectors). This is because different outputs for the connector branches cannot be specified using a single coordinate functional domain. Moreover, the mathematical composition of functions cannot be applied to connector branches. Furthermore, the simulation execution is represented statically based on the component position in the composition. We provide a major improvement by representing components dynamically as functions of states over *time*. Our novel

formalism allows for complex models to be validated as we will show below. Furthermore, simulation execution is represented as a time ordered schedule of component executions. This allows for a more accurate validation process in which the composition execution through time is evaluated.

7.1 Time-based Formalism

In this section we define components, simulation, and validity in the context of our formal validation process. To facilitate the proposed validation process we separately represent a simulation component using our proposed time-based formalism. The separation of the component specification from the component implementation is widely recognized in the simulation community as an important step towards simulation composability and model reuse [88].

7.1.1 Definitions

In our proposed approach, a simulation component is represented as a function of states over time.

Definition 12 (Simulation Component). The formal representation of a simulation component C_i is a function $f_i : X_i \rightarrow Y_i$, where $X_i = I_i \times S_i \times T_i$, and $Y_i = O_i \times S_i \times T_i$. I_i and O_i are the set of input/output messages, S_i is the set of states and T_i is the set of simulation time points at which the component changes state.

By representing a simulation component as a mathematical function we leverage on Petty and Weisel's formal theory of composability [88]. However, our approach greatly improves this by including *time* and *state* as domain coordinates. Our three coordinate representation allows for a meaningful and detailed definition of a valid model without affecting the complexity of the validation process. The domain of each functional

representation is $X_i = I_i \times S_i \times T_i$. Coordinate I_i represents semantically rich inputs, enriched by our COSMO ontology [125]. Next, S_i represents all possible component states. A component state contains all values of the component attributes. Lastly, T_i represents the set of simulation time moments at which state transitions occur.

Definition 13 (Composition). Given the components C_i, $i = \overline{1,n}$ formally represented as f_i. The formal representation of the composed model made up of C_i is $M = \{(f_i, f_j) | i \neq j, i, j = \overline{1,n}\}$ with $(f_i, f_j) \in M$ meaning that C_i is connected to C_j in the composed model with C_j requiring input from C_i.

Definition 14 (Mathematical Composability). Given a composed model $M = \{(f_i, f_j) | i \neq j, i, j = \overline{1,n}\}$, and the time values when f_i produces output and f_j requires input, $T_i^{out} = \{t_m^{(i)} | 1 \leq m \leq |O_i|\}$, and $T_j^{in} = \{t_n^{(j)} | 1 \leq n \leq |I_j|\}$ respectively. Then f_i and f_j are composable iff there exists the bijective binary relation $R = \{(t_n^{(j)}, t_m^{(i)}) \in T_j^{in} \times T_i^{out} | t_n^{(j)} > t_m^{(j)}\}$.

Informally, for the component functions to be composable, all sampled time values for components requiring input must be greater than the time moment values for the components that provide them with output. Definition 14 is the usual mathematical composability definition that only considers the time moment values from the three coordinate function domain. This is because individual component states are irrelevant at this point in the validation, and input and output data has been previously validated as shown in Chapter 6.

A simulation represents the execution of the composition over the simulation time.

Definition 15 (Simulation). The simulation $\mathbb{S}(M)$ of the composed model $M = \{(f_i, f_j) | i \neq j, i, j = \overline{1,n}\}$ is defined formally as the ordered set $\mathbb{S}(M) = \{[\ldots f_i(I_p^i, S_p^i, t_p^i) \rightarrow (O_p^i, S_{p+1}^i, t_{p+1}^i), \ldots, f_j(I_q^j, S_q^j, t_q^j) \rightarrow (O_q^j, S_{q+1}^j, t_{q+1}^j) \ldots] | t_p^i \leq t_q^j, t_p^i \leq t_{p+1}^i, t_q^j \leq t_{q+1}^j, i, j \in \overline{1,n}, \}$ where $I_p^i \in I_i$, $S_p^i, S_{p+1}^i \in S_i$, $t_p^i, t_{p+1}^i \in T_i$, and $I_q^j, \in I_j$, $S_q^j, S_{q+1}^j \in S_j$, $t_q^j, t_{q+1}^j \in T_j, O_p^i \in O^i, O_q^j \in O^j$..

Informally, the simulation $\mathbb{S}$ of a composition of functions M is defined as the ordered set of the function executions for all components. The set order is based on the time t_p^i at which each function f_i is executed. For example, if $(f_i, f_j) \in M$, then at least one function execution of f_i will come before all function executions of f_j. Thus, the simulation of the composed model is an ordered schedule of the function executions. This provides an accurate representation of the simulation to facilitate the validation process. Previous approaches such as that of Petty and Weisel [87] do not consider *time* in the simulation representation. The Petty and Weisel approach employs a *static* simulation description in which components appear in the linear order of aggregation in the composed model. Using our proposed time-based formalism, we obtain a *dynamic* representation of the simulation, in which components appear based on the time moments when they run.

7.1.2 Validation Process

The proposed formal validation approach aims to formally compare between the simulation of the composed model and the simulation of a reference model. The reference model is defined below.

Definition 16 (Reference Model). Given a composed model of components C_i represented formally as $M = \{(f_i, f_j) | i \neq j, i, j = \overline{1, n}\}$, the reference model is defined as $M^* = \{(f_i^*, f_j^*) | i \neq j, i, j = \overline{1, n}\}$, where C_i^* formally represented as f_i^* is the corresponding reference component for component C_i.

To facilitate the comparison between the composed model simulation, $\mathbb{S}(M)$, and the reference model simulation, $\mathbb{S}(M^*)$, the two simulations are represented as Labeled Transition Systems (LTS) [114]. Next, we compare the two LTS using the well-established theory of bisimulation [86].

Definition 17 (Simulation Representation). Given a composed model M and its simulation $\mathbb{S}(\mathbb{M})$. The simulation run $\mathbb{S}$ is represented as a Labeled Transition System $L(M) = (N, Act, \rightarrow)$ where N is the set of nodes, $\rightarrow$ is the set of transitions between nodes, and Act is the set of transition labels. In $L(M)$, each node in N represents an annotated composition state given by the tuple $S_{j=\overline{1,r}} = [\{state(C_i)_{i=\overline{1,n}}\}, f_{in}, f_{out}]$, where $state(C_i)$ is the state of component C_i, $r = |\mathbb{S}|$, n is the number of components, f_{in} is the function called to enter this node, and f_{out} is the function called to exit this node. Edges $\rightarrow$ are the function calls f_{in} or f_{out} in the simulation run, and labels $a_i \in Act$ are the tuple <*function_name*(f_{out}), *duration*(f_{out}), *output*(f_{out})>, where $duration(f_{out})$ represents the execution time of f_{out}.

A simulation run is represented as an LTS where nodes represent the entire composition state as a reunion of the individual component states, and edges are labeled to facilitate the validation process. To facilitate accurate comparison between $L(M)$ and the reference LTS $L(M^*)$, the edge labels contain the name of the function called to exit the node, its duration, and its output. We consider the $duration$ rather than the $time$ moment when f_{out} begins to execute, because the time moments at which the functions f_{out} start to execute are already ordered through the directed nature of simulation $\mathbb{S}$.

Based on the definitions presented above, we refine the formal validation process to the five steps presented in Figure 7.1. The first three steps of the validation process, namely *Unfolding and Sampling*, *Composition*, and *Simulation* are applied separately to the components and reference components. Components and reference components annotated with a star symbol (*) from the composition and reference composition respectively are formally represented as functions of their states over time according to Definition 12. The formal component representations are input to the *Unfolding and Sampling* step where the component representation is adjusted to fit our validation process. Based on the composed model topology, the unfolded representations obtained

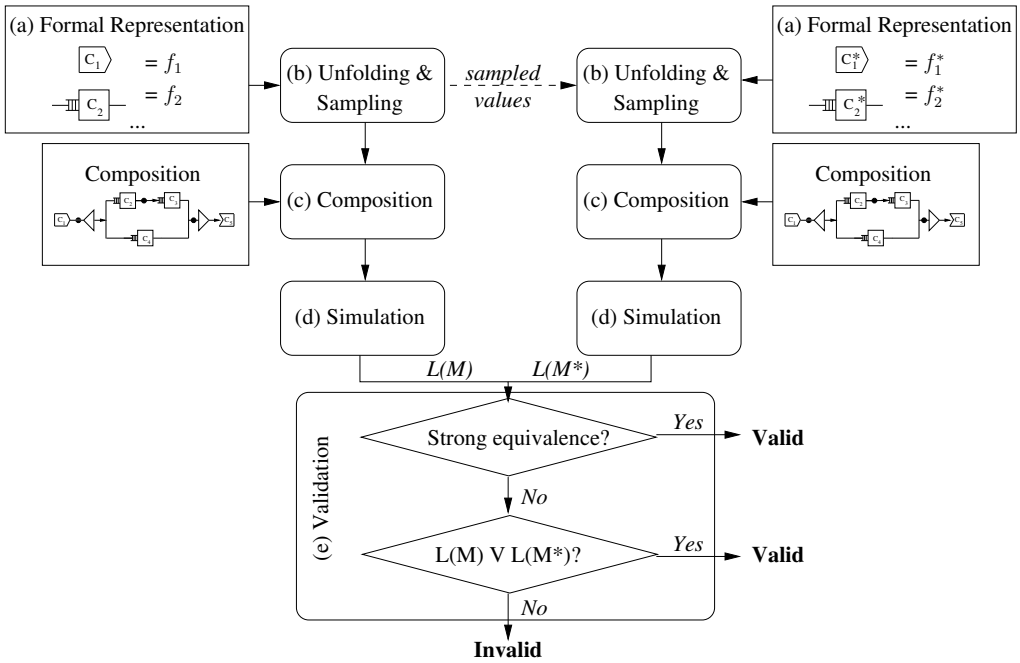


Figure 7.1: Formal Validation Process

from the *Unfolding and Sampling* step are composed according to Definition 14 in the *Composition* step. The *Simulation* step applied to the composition and reference composition results in a composition simulation, $L(M)$, and reference composition simulation, $L(M^*)$, respectively according to Definition 17. The *Composition* step formally composes the functional representations based on our mathematical composability definition, which considers the time moments at which the functions are activated. As such, $L(M)$ and $L(M^*)$ consist of time-ordered simulation schedules of the function executions. Lastly, in the *Validation* step, we first attempt to determine whether $L(M)$ and $L(M^*)$ are exact matches. This is done by determining strong equivalence between $L(M)$ and $L(M^*)$. If strong equivalence is not possible, we introduce the semantic relation V_ϵ to determine weak equivalence only between related states, i.e., the parts in the two executions that are semantically related. If V_ϵ is not a weak bisimulation relation between $L(M)$ and $L(M^*)$, then the model is invalid.

Assumptions For the formal validation process to be realized, a series of assumptions must be in place. Firstly, component information including state machine description must be available in the form of a meta-component in which component attributes and state machine are defined in a standardized format. The component creator provides the component state machine when the component is added to the component repository. For example, components in our CoDES framework are represented as a meta-component in a standardized COML format [125]. Next, logical composition properties such as safety and liveness [82] over time have been previously validated. This is the case in the layered validation process in the CoDES framework in which logical properties are first validated in the context of instantaneous transitions and secondly, over time [119]. Lastly, to facilitate the calculation of V_ϵ, a component-based ontology in which component-based simulation concepts as well as application domain notions are rigorously defined. In the CoDES framework, the COSMO ontology describes component-oriented simulation within and across application domains [125].

7.1.3 Validity Measures

In our proposed formal theory, we consider two possible relations between the simulation of the composed model and the simulation of the reference model, $L(M)$ and $L(M^*)$ respectively: strong equivalence relation [86] and our proposed semantic parametric metric relation, V_ϵ. Informally, strong equivalence between $L(M)$ and $L(M^*)$ validates that $L(M)$ is exactly the same or included in $L(M^*)$, including the sequence of the function calls and the edge labels. If this is not possible, we propose the semantic parametric relation V_ϵ as a weak bisimulation relation. V_ϵ considers only parts of $L(M)$ and $L(M^*)$ that are semantically close and validates that they appear in the same sequence in $L(M)$ and $L(M^*)$. V_ϵ is defined below.

Definition 18 (Semantic Parametric Metric Relation). Let $P \subseteq \{S_1, \ldots, S_n\}$, $Q \subseteq$

$\{S_1^*, \ldots, S_n^*\}$ a subset of the annotated composition states for $L(M)$ and $L(M^*)$ respectively, with $p \in P$, $q \in Q$, $p = [s(p), f_{in}(p), f_{out}(p)]$, $q = [s^*(q), f_{in}^*(q), f_{out}^*(q)]$, with $s(p) = [state(C_1), \ldots, state(C_n)]$ and $s^*(q) = [state(C_1^*), \ldots, state(C_n^*)]$ vectors representing component states. We define the semantic relation with parameter ϵ, $V_\epsilon \subseteq P \times Q$, as $V(p,q) = \{(p,q) \in P \times Q | \|p - q\|_\sigma \leq \epsilon\}$. The semantic vector norm, $\|p - q\|_\sigma$, is defined as

$$\|p - q\|_\sigma = \frac{DS(s(p), s^*(q)) + \frac{DF(f_{in}(p), f_{in}^*(q)) + DF(f_{out}(p), f_{out}^*(q))}{2}}{2}$$

where $DS(s(p), s^*(q))$ is the semantic distance between composition states, and $DF(f_i, f_j^*)$ is the semantic functional distance between the function names.

The semantic metric relation with parameter ϵ, V_ϵ, contains semantically related states between $L(M)$ and $L(M^*)$. Semantically related states are those for which the semantic vector norm, $\| \quad \|_\sigma$, is smaller than the parameter ϵ. The semantic vector norm has two components, DS and DF. The semantic state distance, DS, measures the semantic differences between component attribute values. The semantic functional distance, DF determines whether the functions that are called to enter and exit the LTS nodes are related.

Definition 19 (Semantic State Distance). Let $s(p) = [state(C_1), \ldots, state(C_n)]$, $s^*(q) = [state(C_1^*), \ldots, state(C_n^*)]$. The semantic state distance between vectors p and q is defined as

$$DS(s(p), s^*(q)) = \frac{\sum_{i=1}^{n} |ds(state(C_i), state(C_i^*))|}{n}$$

where $ds(state(C_i), state(C_i^*)) = \frac{\sum_{a_i \in A(C_i), a_j^* \in A(C_j^*)} d(a_i, a_j^*)}{m}$, $A(C_i)$ is the set of at-

tributes for component C_i, $m = |A(C_i)|$ and $d(a_i, a_j^*)$ is defined as

$$d(a_i, a_j^*) = \begin{cases} 0 & \text{if related}(a_i, a_j^*) \text{ and value}(a_i) = \text{value}(a_j^*) \\ 0.5 & \text{if related}(a_i, a_j^*) \text{ and value}(a_i) \neq \text{value}(a_j^*) \\ 1 & \text{if } \nexists a_j^* \in A(C_i^*) \text{ such that related}(a_i, a_j^*) = \text{true} \end{cases}$$

where $related(a_i, a_j)$ signifies that a_i and a_j are related in the COSMO ontology.

Definition 20 (Semantic Function Distance). Let $f_i(p), f_j^*(q)$ be the functions called to enter or exit nodes p and q in $L(M)$ and $L(M^*)$ respectively. The semantic state distance DF is defined as

$$DF(f_i(p), f_j^*(q)) = \begin{cases} 1, & i \neq j \\ 0, & i = j \end{cases}$$

The calculation of the semantic distance DS is facilitated by the COSMO ontology. This is done by determining the similarity between component states (ds) by calculating the semantic closeness in the ontology of all component attributes (d). Informally, V_ϵ determines whether semantically related states from $L(M)$ and $L(M^*)$ (in terms of composition state - DS, and incoming and outgoing function calls - DF) appear in the same labeled sequence in $L(M)$ and $L(M^*)$ respectively. The above definition is similar to that of Petty and Weisel [88]. However, the fundamental difference and our major improvement comes from forcing the weak bisimulation relation to be V_ϵ which we previously defined. V_ϵ is a *semantic* metric relation that considers related composition states according to the COSMO ontology in which a well-defined component and attribute hierarchy is present. By representing components as functions of times and states, $L(M)$ and $L(M^*)$ can be compared based on the timed sequences of component

executions. Through V_ϵ, the model can be compared with a reference model based on rigorously defined concepts in an ontology.

Definition 21 (Validity). Given the composed model $M = \{(f_i, f_j)|i \neq j, i, j = \overline{1,n}\}$ and its simulation representation $L(M)$, and a reference model $M^*\{(f_i^*, f_j^*)|i \neq j, i, j = \overline{1,n}\}$ with the simulation representation $L(M^*)$. M is valid iff $(f_i, f_j) \in M$ and $(f_i^*, f_j^*) \in M^*$ are composable respectively (by Definition 14) and there exists a binary relation R between $L(M)$ and $L(M^*)$, with $L(M) \quad R \quad L(M^*)$ such that R is either a strong equivalence relation [86] or a weak semantic parametric relation, V_ϵ.

7.2 Theoretical and Experimental Analysis

7.2.1 Theoretical Analysis

The time complexity of this layer, O_{PM}, is calculated as the sum of three main parts:

$$O_{PM} = O_{transform} + O_{compose} + O_{bisimulate} \tag{7.1}$$

where $O_{transform}$ is the time complexity for the formal component representation, unfolding and sampling, and simulation steps (Step 1, 2, and 4); $O_{compose}$ is the time complexity for the Composition step (Step 3) and $O_{bisimulate}$ is the time complexity for the Validation Step (Step 5). The time complexity for the formal component representation and the unfolding and sampling steps, as well as the simulation step is in the worst case $O(n)$. Thus,

$$O_{transform} = O(n) \tag{7.2}$$

The time complexity of the Composition step is reduced to the time complexity required by a constraint solver implementation to solve the proposed constraints. The constraint

satisfaction problem is NP-Complete. However, the algorithm that solves the particular set of constraints from the Composition step has the time complexity of $O(n)$. This is because we require a single solution, which can be obtained by fixing the values for the time moments for the source components (e.g. x in Equation 7.6) and propagating the values to the rest of the variables. Therefore,

$$O_{compose} = O(n) \tag{7.3}$$

For two LTS with N nodes and M transitions, strong and weak bisimilarity between two states can be determined in $O(MN)$ [60]. As such, strong and weak bisimilarity between two LTS can be determined in $O(N^2M)$. For the two LTS that are obtained in the reference Model Validation, we have $N = \tau n$ and $M = N - 1 = \tau n - 1$. Thus,

$$O_{bisimulate} = O(\tau^2 \times n^2 \times (\tau n - 1)) = O(n^3) \tag{7.4}$$

Combining Equations 7.2, 7.3, and 7.4, Equation 7.1 becomes:

$$O_{PM} = O(n) + O(n) + O(n^3) = O(n^3) \tag{7.5}$$

Therefore, the complexity of the reference Model Validation is *polynomial* in the number of components.

Given n, the number of components in the composition and s, the maximum number of messages allowed in the connectors, we summarize our results in Table 7.1.

Layer	Complexity
Concurrent Process Validation	$O(s^{2P^1(n)} \times \prod_{i=1}^{n} T_i)$
Meta-Simulation Validation	$O(2^{n \times P^1(n)})$
reference Model Validation	$O(n^3)$

Table 7.1: Theoretical Analysis of the Validation Process

7.2.2 Experimental Analysis

We demonstrate our approach using a simple single-server queue example presented in Figure 4.7. Each component has an attached implementation as described in the meta-component [125]. Key meta-component information is the same as the one shown in Table 6.1 in the previous chapter. To focus on our approach, we consider only simplified component state machines.

Validation of General Model Properties

As presented in Chapter 6, the first step of our proposed dual-step validation process validates the composed model for general model properties, which include component communication, and component coordination in the context of instantaneous (Concurrent Process Validation) and timed (Meta-simulation Validation) transitions. We assume that the model in Figure 4.7 has passed the validation of general model properties as shown in Section 6.4.2.

Validation of Model Execution for a Single-Server Queue Model

In the following we present the detailed validation process only for the selected components C_i represented formally as functions f_i. The same process is repeated for reference functions f_i^*. For this example, we consider the behavior of the reference com-

ponents represented by f_i^* to be the same with respect to input/output transformations to the behavior represented by f_i. The base components C_i differ from the reference components C_i^* through additional logging attributes such as $noJobsGenerated$.

Formal Component Representation

In the *Formal Component Representation* step, the state machine for component C_1 as specified in its meta-component is $S_1(\Delta interArrivalTime) \rightarrow S_2, S_2 \rightarrow S_1O_1[A_1]$. This expression is translated to a formal component representation specified by f_1 which is defined as

$$f_1 : \emptyset \times S_1 \times T_1 \rightarrow \{O_1\} \times S_1 \times T_1, f_1(\emptyset, s_i, t) \rightarrow (O_1, s_i', t + \Delta t)$$

where Δt is sampled from a specified distribution and the function is re-called until $t > T$, where the simulation runs for time $T = 40$ wall clock units.

Unfolding and Sampling

The above expression is not useful for the *Unfolding and Sampling* step in our approach since during a simulation run, t and Δt have specific values. Thus we unfold the function call graph for $\tau = 3$ times and sample the values for Δt, using mean values provided by the user. For component C_1 assume that the inter-arrival time is sampled from an exponential distribution with a $mean = 3$. With sampling and an unfolding grade of $\tau = 3$ we have $\Delta t = 6, \Delta t = 2, \Delta t = 4$. For component C_2 described formally by f_2 assume the service time has an exponential distribution with a mean of $mean = 6$ sampled as $11, 6, 1$. Lastly, we assume component C_3 formalized in f_3 takes 1 unit of time to service each job, so $\Delta t = 1$ for all samples. The values of f_1, f_2, and f_3 are presented in Table 7.2.

	Unfold	Δt	**Formula**
f_1	1	6	$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
	2	2	$f_1(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
	3	4	$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
f_2	1	11	$f_2(I_2, s_1^2, x \geq 0) \rightarrow (O_2, s_2^2, x + 11)$
	2	6	$f_2(I_2, s_2^2, t \geq x + 11) \rightarrow (O_2, s_3^2, t + 6)$
	3	1	$f_2(I_2, s_3^2, r \geq t + 6) \rightarrow (O_2, s_4^2, r + 1)$
f_3	1	1	$f_3(I_3, s_1^3, x' \geq 0) \rightarrow (\emptyset, s_2^3, x' + 1)$
	2	1	$f_3(I_3, s_2^3, t' \geq x' + 1) \rightarrow (\emptyset, s_3^3, t' + 1)$
	3	1	$f_3(I_3, s_3^3, r' \geq t' + 1) \rightarrow (\emptyset, s_4^3, r' + 1)$

Table 7.2: Formal Component Representation

Composition

Next, the function composability is validated in the *Composition* step. Following Definition 14 we obtain constraints for the values of x, t, r and x', t', r' respectively. The constraints on x, t, r are derived from the fact that the first call to function f_2 has to take place after at least one call to f_1 has completed and produced output, since f_2 requires output from f_1. Similarly for f_3, the first call has to take place after f_2 has produced *at least one* output. Furthermore, the average time spent by messages in the connector queues is considered. The average time in queue is obtained from the meta-simulation validation layer. Assuming that the average times spent in the connector queues are $\Delta w_1 = 2, \Delta w_2 = 3, \Delta w_3 = 1$ and $\Delta w_1' = 4, \Delta w_2' = 3, \Delta w_3' = 2$ for f_2 and f_3 respectively, the most trivial constraints that can be derived are:

$$x \geq 6 + \Delta w_1, t \geq x + 11, t \geq 8 + \Delta w_2, r \geq t + 6, r \geq 12 + \Delta w_3 \tag{7.6}$$

$$x' \geq x + 11 + \Delta w_1', t' \geq x' + 1, t' \geq t + 6 + \Delta w_2', r' \geq t' + 1, r' \geq r + 1 + \Delta w_3' \tag{7.7}$$

Next, the constraints are solved by a constraint solver [1]. Assume that a solution is:

$$(x = 8, t = 19, r = 25), (x' = 23, t' = 28, r' = 29).$$

[1] We employ the Choco constraint solver [23].

For the reference functions f_i^*, the constraint solver returns the same solution for (x^*, t^*, r^*) and (x'^*, t'^*, r'^*):

$$(x^* = 8, t^* = 19, r^* = 25), (x'^* = 23, t'^* = 28, r'^* = 29).$$

Simulation

The above solutions dictate the interleaved execution schedules of the function calls. Interleaved execution schedules are obtained for both composition and reference composition. For this simple model in which the component definitions are similar with the exception of some attributes, the interleaved schedules shown in Figure 7.2(a) and Figure 7.2(b) are the same. Each interleaved execution is represented as a Labeled

$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(a) Composition

$f_1^*(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1^*(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2^*(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1^*(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2^*(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3^*(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2^*(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3^*(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3^*(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(b) Reference Composition

Figure 7.2: Interleaved Execution Schedules

Transition System, $L(M)$ and $L(M^*)$ respectively, as shown in Figure 7.3.

Validation

In the *Validation* step, *strong* equivalence between $L(M)$ and $L(M^*)$ is validated using the BISIMULATOR equivalence checker, part of the CADP toolset. For this simple example, the BISIMULATOR returns `true`. As such, there is no need to validate a possible *weak* equivalence by calculating the semantic metric relation V_ϵ.

For a more complex example, we examine below the validation of a single-server queue model in which the source generates two job classes.

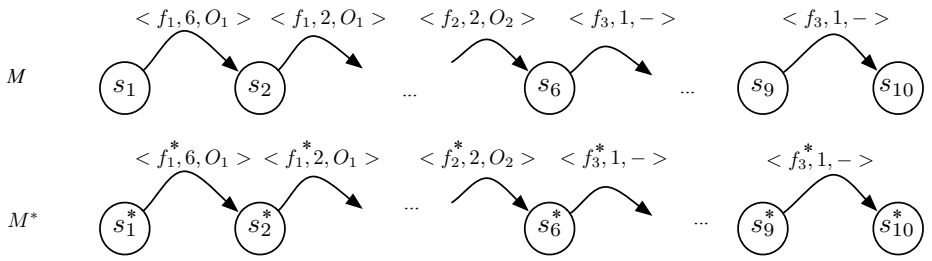


Figure 7.3: LTS Representation of Model Execution

Validation of Model Execution for a Single-Server Queue with Two Job Classes For a more complex example, assume that the composition to be validated represents a single-server queue in which the *Source* generates alternatively two classes of jobs that have different service times when serviced by the *Server* component. The meta-component information is presented in Table 7.3. Assume that the reference components are the same as in the previous example.

Formal Component Representation & Unfolding and Sampling

Following the *Unfolding and Sampling*, step we obtain the formal component representation presented in Table 7.4.

Composition

Similar to the previous example, the function composability is validated in the *Composition* step. Following Definition 14, we obtain constraints for the values of x, t, r and x', t', r' respectively. The constraints on x, t, r derive from the fact that the first call to function f_2 has to take place after at least one call to f_1 has completed and produced output, since f_2 requires output from f_1. Similarly for f_3, the first call has to take place after f_2 has produced *at least one* output. Furthermore, the average time spent by messages in the connector queues is considered. The average time in queue is obtained from the meta-simulation validation layer. Assuming that the average times spent in the

	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$
Attribute	$noJobsGenerated = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ interArrivalTime: exponential(3) $transient(C_1) : (noJobsGenerated == 1)$	$noJobsServiced = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ serviceTime1 : exponential(6) serviceTime2 : exponential(3) $busy = false$ $transient(C_2) : (busy == true)$	$noJobsPrinted = 0$ $timeout = 20$ $time = 200$ $timeScale = 1/5$ $\Delta printingTime = 1$ $transient(C_3) : (noJobsPrinted == 1)$
Input	-	I_1, constraints: $origin = Source\|Server$ $range = 10; 35$ $type = double$ $class = C_1$	I_1, constraints: $origin = Server$
	-	I_2, constraints: $origin = Source\|Server$ $range = 10; 35$ $type = double$ $class = C_2$	-
Output	O_1, constraints: $destination = Server$ $range = 11; 15$ $type = int$ $class = C_1$	O_1, constraints: $destination = Server\|Sink$ $range = 10; 20$ $type = double$	-
	O_2, constraints: $destination = Server$ $range = 11; 15$ $type = int$ $class = C_2$	-	-
State Machine	$S_1(\Delta interArrivalTime) \xrightarrow{[C_1]} S_1O_1[A_1]$ $S_1(\Delta interArrivalTime) \xrightarrow{[C_2]} S_1O_2[A_2]$	$I_1S_1 \rightarrow S_2[A_1; A_3; A_4]$ $I_2S_1 \rightarrow S_2[A_1; A_3; A_5]$ $S_2(\Delta serviceTime1) \xrightarrow{C_1} S_1O_1[A_2]$ $S_2(\Delta serviceTime2) \xrightarrow{C_2} S_1O_1[A_2]$	$I_1S_1 \rightarrow S_2$ $S_2(\Delta printingTime) \rightarrow S_1[A_1]$
	$[A_1] = noJobsGenerated++;$ $[C_1] = noJobsGenerated\%2 == 0;$ $[C_2] = noJobsGenerated\%2 == 1;$	$[A_1] = (busy = true);$ $[A_2] = (busy = false);$ $[A_3] = noJobsServiced++;$ $[A_3] = class = C_1;$ $[A_4] = class = C_2;$ $[C_1] = class == C_1;$ $[C_2] = class == C_2;$	$[A_1] = noJobsPrinted++;$

Table 7.3: Meta-component Information

	Unfold	Δt	**Formula**
f_1	1	6	$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
	2	2	$f_1(\emptyset, s_2^1, 6) \rightarrow (O_2, s_3^1, 8)$
	3	4	$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
f_2	1	11	$f_2(I_1, s_1^2, x \geq 0) \rightarrow (O_2, s_2^2, x + 11)$
	2	2	$f_2(I_2, s_2^2, t \geq x + 11) \rightarrow (O_2, s_3^2, t + 2)$
	3	1	$f_2(I_1, s_3^2, r \geq t + 6) \rightarrow (O_2, s_4^2, r + 1)$
f_3	1	1	$f_3(I_3, s_1^3, x' \geq 0) \rightarrow (\emptyset, s_2^3, x' + 1)$
	2	1	$f_3(I_3, s_2^3, t' \geq x' + 1) \rightarrow (\emptyset, s_3^3, t' + 1)$
	3	1	$f_3(I_3, s_3^3, r' \geq t' + 1) \rightarrow (\emptyset, s_4^3, r' + 1)$

Table 7.4: Formal Component Representation

connector queues are $\Delta w_1 = 2, \Delta w_2 = 3, \Delta w_3 = 1$ and $\Delta w'_1 = 4, \Delta w'_2 = 3, \Delta w'_3 = 2$ for f_2 and f_3 respectively, the most trivial constraints that can be derived are:

$$x \geq 6 + \Delta w_1, t \geq x + 11, t \geq 8 + \Delta w_2, r \geq t + 2, r \geq 12 + \Delta w_3 \tag{7.8}$$

$$x' \geq x + 11 + \Delta w'_1, t' \geq x' + 1, t' \geq t + 2 + \Delta w'_2, r' \geq t' + 1, r' \geq r + 1 + \Delta w'_3 \tag{7.9}$$

Next, the constraints are solved by a constraint solver. Assume that a solution is:

$$(x = 8, t = 19, r = 21), (x' = 23, t' = 24, r' = 25).$$

For the reference functions f_i^*, the constraint solver returns the same solution for (x^*, t^*, r^*) and (x'^*, t'^*, r'^*):

$$(x^* = 8, t^* = 19, r^* = 25), (x'^* = 23, t'^* = 28, r'^* = 29).$$

Simulation

The above solutions dictate the interleaved execution schedules of the function calls. Interleaved execution schedules are obtained for both composition and reference composition, as shown in Figure 7.4(a) and Figure 7.4(b) respectively. Each interleaved ex-

$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1(\emptyset, s_2^1, 6) \rightarrow (O_2, s_3^1, 8)$
$f_2(I_1, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 22)$
$f_2(I_1, s_3^2, 22) \rightarrow (O_2, s_4^2, 23)$
$f_3(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_3(I_3, s_2^3, 24) \rightarrow (\emptyset, s_3^3, 25)$
$f_3(I_3, s_3^3, 25) \rightarrow (\emptyset, s_4^3, 26)$

(a) Composition

$f_1^*(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1^*(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2^*(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1^*(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2^*(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3^*(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2^*(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3^*(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3^*(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(b) Reference Composition

Figure 7.4: Interleaved Execution Schedules

ecution is represented as a Labeled Transition System, $L(M)$ and $L(M^*)$ respectively,

as shown in Figure 7.5.

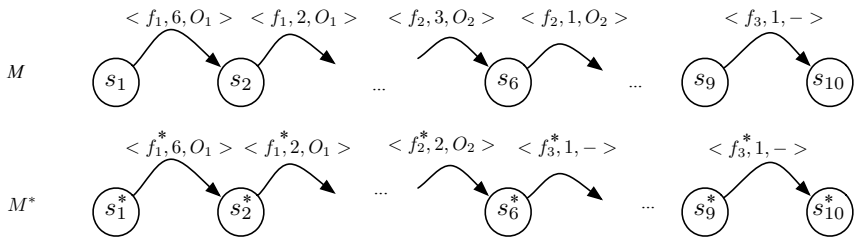


Figure 7.5: LTS Representation of Model Execution

Validation

It is evident that the two LTS are not strongly equivalent (see the outgoing labels from S_6 and S_6^*), hence the BISIMULATOR tool returns `false`. We calculate the semantic metric relation V_ϵ for $\epsilon = 0.5$ and obtain the following related nodes: $V_\epsilon = \{(S_1, S_1^*)$, (S_2, S_2^*), (S_3, S_3^*), (S_4, S_4^*), (S_3, S_5^*), (S_5, S_5^*), (S_7, S_6^*), (S_7, S_8^*), (S_9, S_9^*), (S_{10}, S_9^*), $(S_{10}, S_{10}^*)\}$, with $\{\|S_i - S_j^*\|_\sigma = 0.41 | \forall (i,j) \neq (5,5)\}$ and $\{\|S_5 - S_5^*\|_\sigma = 0.45\}$. For these values of V_ϵ we can conclude that the model is not valid since V_ϵ is not a weak bisimulation relation between $L(M_1)$ and $L(M^*)$.

Discussion

The above examples raise some interesting issues. Firstly, there is the well-known difference between what system experts perceive as *valid* and what can be defined in a computer system as a *valid model* for it to validate automatically and independently. In our formal approach, a valid model is one that is *close enough* with respect to the states, sequence and duration of component execution, to a reference model. Yet, what exactly is close enough (i.e., the values of ϵ), as with all thresholds, remains an open problem. For the examples presented in this section, the value of ϵ has been set to $\epsilon = 0.5$ after a series of experiments that looked at various valid and invalid composed models. Our initial experiments seem to suggest that the value of ϵ depends mainly on the structure

of the generic reference components, which we can say that depends on the application domain. For example, if the reference components have a lot of specific attributes and a state machine behavior with a large number of states, this will result in a high number of nodes in $L(M^*)$, for which a lower value of ϵ might be more suited.

Our second experiment for the validation of a single-server queue with two classes of jobs showed another limitation of our approach, which stems from the hidden assumptions that can appear in the reference components. In particular, the current reference components had a single time attribute that needed to be sampled. This did not work well for the models with two classes of jobs because different service times were necessary for each individual job class. For example, the single-server queue with two classes of jobs described above would be considered valid if the reference model for the Server component would contain two sampling time intervals instead of a single one. This will be the inherent problem when creating reference generic components specific to each application domain.

The problem of reference models remains. While it is acceptable to assume their existence, their origin (i.e., who provides them) and content (i.e., timed attributes, internal structure) is still an open question. Previous approaches such as Petty and Weisel's [88] do not consider the nature of the reference components. This is the first time that the exact structure of the reference models has been studied.

Lastly, the impact of a different semantic distance DS on the weak semantic bisimulation relation remains to be studied.

7.3 Summary

Thorough study of semantic composability validation shows that in the literature there are various degrees of model validity and validity is not a yes/no answer. Current approaches to validate composed models are theoretically elegant but do not apply to

complex simulation models [88] or are computationally expensive and do not scale [127, 77]. Based on our composability studies of reusing base components, we observe that there are more invalid than valid models. Moreover, checking for absolute validity is more expensive than if we employ a dual-step deny validity approach. Firstly, invalid models are eliminated by validating general model properties for both instantaneous and timed transitions. If the model passes this test, formal model execution validation is performed. Table 7.5 presents a summary of our validation approach.

What	How	Outcome
1. Model Properties	- Component Communication - Concurrent Process Validation - Meta-simulation Validation	- Communication is semantically aligned - Interleaved execution is correct (timeless). - Interleaved execution is correct (time + other attributes.)
2. Formal Validation	Time-based validation with reference model.	- Quantitative measure of similarity to reference model.

Table 7.5: Summary of Semantic Composability Validation

In this chapter, we propose a formal approach for the validation of semantic composability. We introduce a novel time-based formalism, in which a simulation component is represented as a function of its states over time. Based on our formal definitions of composition, simulation, and validity, we refine and specialize the formal validation process to suit component-based simulations, in which time and state are of paramount importance and the behavior of the composed model over time is compared to the behavior of a reference model. The comparison determines the equivalence of the schedules based on a new semantic metric relation. Our theoretical analysis shows that the validation process has polynomial complexity and our execution time analysis shows that our approach is scalable. We have fully implemented and integrated the validation process in our CoDES component-based framework and tested our formal approach on models of varying size and complexity from two application domains, Queueing Networks, shown in Appendix B, and Military Training Simulations, shown in Appendix D.

Chapter 8

Prototype and Evaluation

In the design of the CoDES framework, three main considerations are: (i) to provide for the reuse and composability of simulation components in an integrated approach with low cost for each simulation life-cycle step, (ii) the accurate semantic validation of the composed model to increase model credibility, and (iii) a component-based framework that is easily implemented. This chapter presents an overview of our implemented prototype of the CoDES framework. Different application domains can co-exist seamlessly in the CoDES framework, with reuse possible both within and across application domains. We demonstrate our approach for two application domains, namely, Queueing Networks and Military Training Simulations.

We show the benefits and inherent trade-offs of our proposed approach as they are observed in two main experiments. In the first experiment, we evaluate the cost of semantic validation of data-driven components in the Military Training Simulations application domain. We identify trade-offs between accuracy and computational cost and evaluate the state space growth and runtime execution of the validation process. Furthermore, to determine the scalability of our deny validity approach, as well as the incremental cost of each step, we evaluate the runtime execution of semantic composability validation for large models in the Queueing Networks application domain. In

the second experiment, we analyze the benefits and cost of model component reuse by composing a grid system in the Queueing Networks application domain. The grid system has sixteen components and is observed from the conceptual stage to the validated model, using base and model components to showcase different types of reuse.

8.1 Prototype Design and Implementation

Figure 8.1 presents an overview of the proposed solutions for the modular design proposed in Chapter 3. The structure of our prototype implementation is according to the four-step life-cycle of component-based modeling and simulation proposed in Chapter 3. The verification of syntactic composability is achieved by the *SyntaxVerifier* module that employs the proposed EBNF compositional grammar. In model discovery and selection, the *ModelLocator* module calculates the Matching Index using the COSMO ontology. Lastly, in semantic composability validation, the *SemanticValidator* module implements our proposed deny-validity approach.

A high level overview of the Java prototype implementation of the CoDES framework is shown in Figure 8.2. The CoDES prototype implementation revolves around the three key modules discussed above: *SyntaxVerifier*, *ModelLocator*, and *SemanticValidator*. All modules rely on the meta-component representation in COML. The COML specification is described in XSD and each meta-component is defined by a XML file. The COSMO ontology is implemented in RDF using the Protege tool. The *SyntaxVerifier* module implements the Earley parser algorithm [37] and relies on an EBNF representation of the CoDES composition grammar. In model discovery and selection by the *ModelLocator* module, we employ the Jena reasoner to query the COSMO ontology and reason about similarity. The *SemanticValidator* module relies on three auxiliary modules for validation: ConcurrentProcessValidation, MetaSimulationValidation, and FormalValidation. *ConcurrentProcessValidation* employs the SPIN model checker

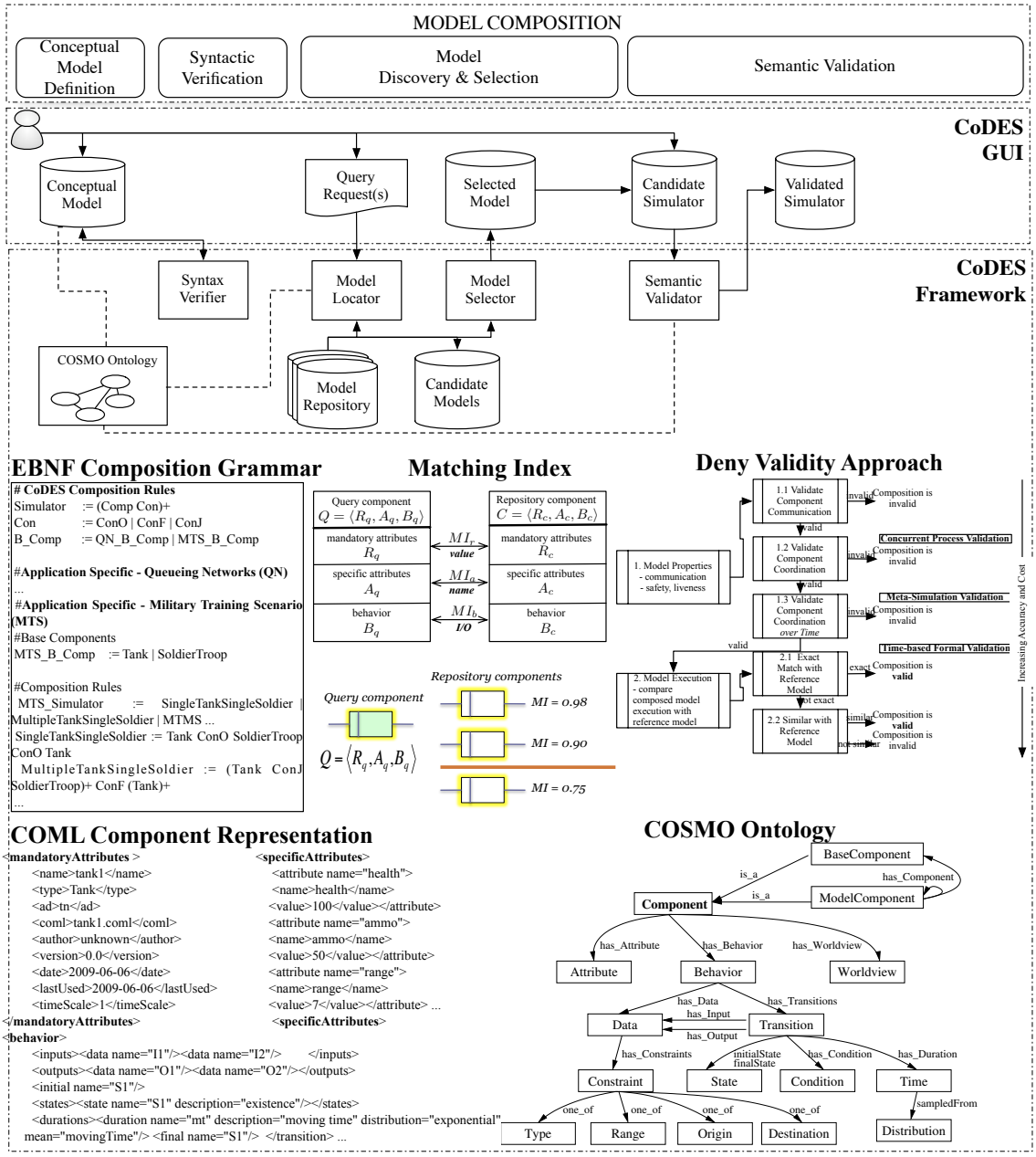


Figure 8.1: Overview of CoDES Prototype Design

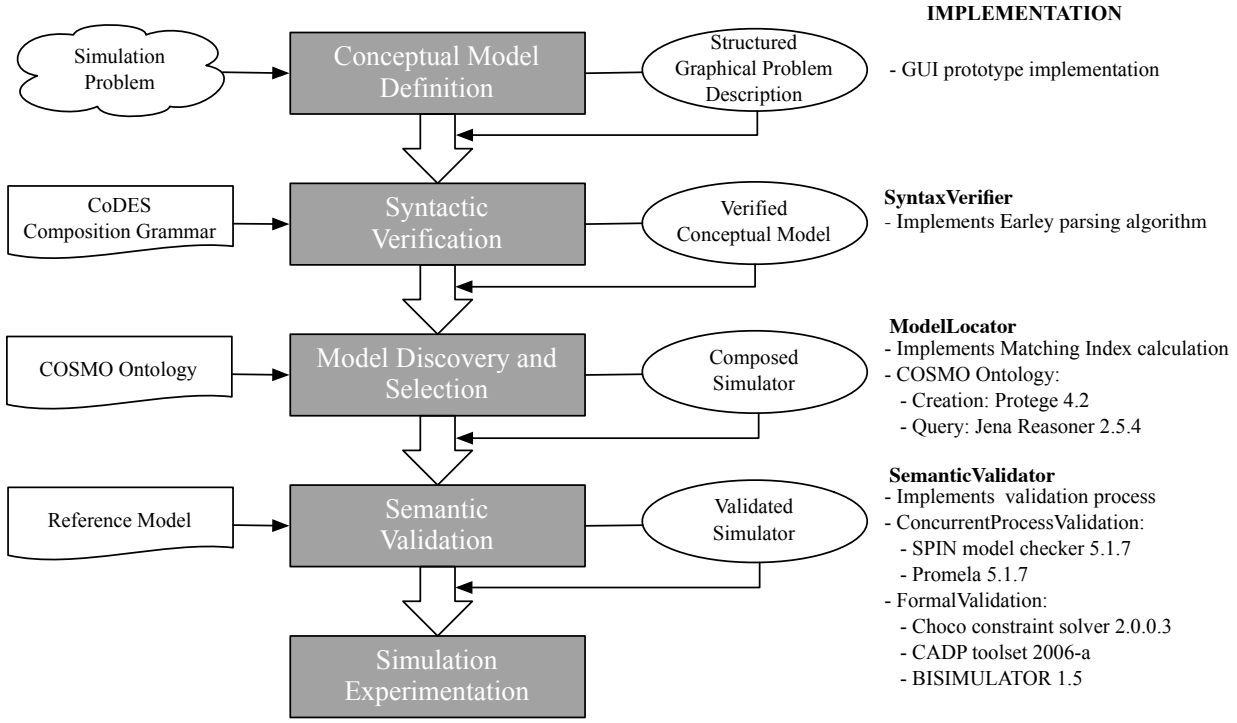


Figure 8.2: CoDES Implementation Overview

to verify a Promela specification of the composed model. *MetaSimulationValidation* translates the composition into a Java class hierarchy that is subsequently executed. Lastly, *FormalValidation* employs the BISIMULATOR tool from the CADP toolset to reason about equivalence with the reference perfect model. Detailed class diagrams are presented in Appendix E.

8.2 Composable Queueing Networks Simulations

To add a new application domain to the CoDES framework involves extending the COSMO ontology with descriptions of the domain base components, including defining hierarchies for attributes and properties. Next, the framework composition grammar is extended by adding composition rules specific to the new application domain. We have presented an overview of the Queueing Networks application domain in Chapter 3.

Next, Chapter 4 has shown how the addition of a new application domain is reflected in the basic COSMO ontology. A diagram of the COSMO ontology for Queueing Networks is shown in Figure 5.4. For any new application domain added to the CoDES framework, a new specific composition grammar must be added to the CoDES Composition Grammar. In this case, composition rules specific to the Queueing Network application domain define the connectivity of the base components (i.e., source, server, sink) to form different queueing network systems using the three types of CoDES connectors, as shown in Chapter 4.

8.3 Composable Military Training Simulations

In a similar manner, we extend the COSMO Ontology and the CoDES composition grammar to include the new Military Training Simulation application domain. For simplicity, we present a Military Training application consisting of two base components, namely, a *Tank* that models a tank unit, and *SoldierTroop* that models a troop of soldiers. The addition of the military training base components to the repository is reflected in the ontology and in the composition grammar, as shown in Figure 8.3 and Figure 8.4 respectively.

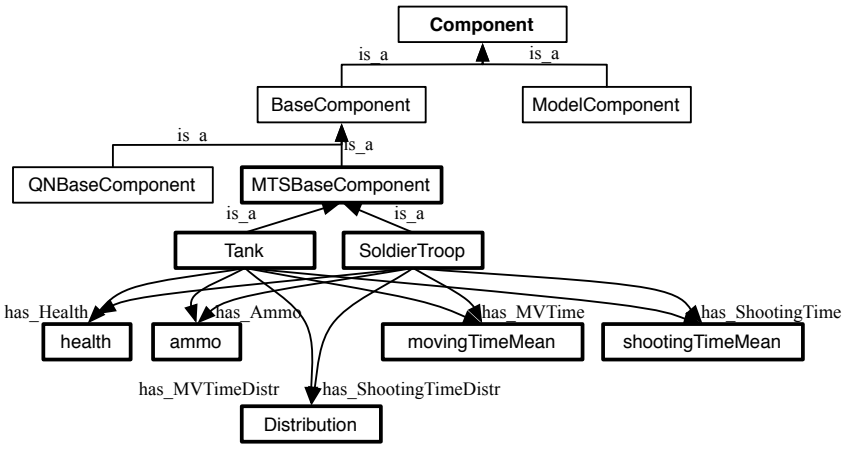


Figure 8.3: Extended COSMO Ontology for Military Training Simulation Domain

MTSBaseComponent is added as a new subclass of the BaseComponent class, with *Tank* and *SoldierTroop* as its subclasses. Besides having attributes and behavior like their superclass, the *Tank* and *SoldierTroop* components must have a *movingTimeMean* attribute and a moving time *Distribution*, and a *shootingTimeMean* and a shooting time *Distribution*, as well as a *health* and an *ammo* attribute. *hasMVTime*, *has_ShootTime*, and *hasMVTimeDistr*, *has_ShootTimeDistr* are subproperties of *has_Attribute* and *has_Distribution* respectively.

```
#  CoDES Reuse Rule
Model_Comp ::= QN_Simulator | MC_Simulator
...
# Application Domain Composition Rules
# Military Training Simulations (MTS)
# Base Components
MTS_B_Comp                  ::= Tank | SoldierTroop

# MTS Composition Rules
MTS_Simulator               ::= SingleTankSingleSoldier |
                                MultipleTankSingleSoldier | MTMS ...
SingleTankSingleSoldier     ::= Tank ConO SoldierTroop ConO Tank
MultipleTankSingleSoldier   ::= (Tank ConJ SoldierTroop)+ ConF (Tank)+
```

Figure 8.4: Composition Grammar for Military Training Application Domain

Since this is the first time we introduce the Military Training application domain in this thesis, we discuss in detail how a composed model for the Military Training application domain is developed. As shown in Figure 8.5, a new simulation model is developed using our graphical input model interface by drag-and-drop icons representing base components of soldiers and tank. The conceptual model is a closed system with feedback loop. The conceptual model is syntactically verified against the new extended composition grammar described in Figure 8.4. Once a valid answer is returned, each base component is discovered as discussed in Chapter 5.

In contrast with the Queueing Networks application domain, where the base components are entities with simple behavior, the components in the Military Training application domain have more complex behaviors and are defined as *data-driven*. The

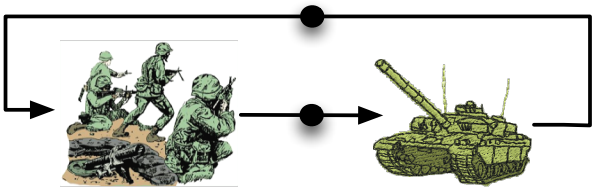


Figure 8.5: Tank vs Soldier Troop Training Simulation

complexity in terms of the number of attributes and elaborate logic has impact on the COML component representation and on the semantic validation process. For example, the state of the Tank component changes dynamically based on the GPS coordinates of its enemy, and many internal attributes such as available ammunition, damage, and attack tactics (e.g. direct charge, shoot and scoot, ambush) [129]. In this respect, the tank component is considered to be *data-driven*. Because of the complex nature of the components, which results in a very large simulation state space, the validation of a composed model from data-driven components is a complicated and lengthy process resulting in increased costs and development time [30]. Let us assume that the components have been discovered according to the process described in Chapter 5 and that their meta-components, $tank_1$ and $troop_1$, are described in Table 8.1.

The combined state machines for the two components is shown informally in Figure 8.6, with full and dashed lines representing transition changes and message exchange, respectively. Both tank and soldier troop have an initial position on a two dimensional grid, a number of ammunition shots, and a speed with which they move. For both components, the moving time and the shooting time are sampled from exponential distributions with various mean values. When a component receives the opponent's position, it will move towards the opponent if the opponent is not in range (condition C_1 and attribute change A_1), or it will otherwise fire if it has enough ammunition and is not severely damaged (condition C_2 and attribute change A_2). When a component is shot

Entity	Attribute	Input	Output	State Machine
$\mathbf{tank_1}$	$health = 100$ $range = 7$ $ammo = 50$ movingTime: exponential(5) shootingTime: exponential(4) $usableThreshold = 20$ $positionX = 20$ $positionY = 15$ $speed = 10$ $team = red$ ... $transient(tank_1) : (ammo == 49)$	I_1, constraints: $class = PositionInfo$ $origin = SoldierTroop$ I_2, constraints: $class = InputFire$ $origin = SoldierTroop$	O_1, constraints: $class = PositionBroadcast$ $destination = SoldierTroop$ O_2, constraints: $class = OutputFire$ $destination = SoldierTroop$	$I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_2A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_2A_4$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_1S_1 \xrightarrow{C_3} O_1S_1$ $null \quad S_2(\Delta movingTime) \rightarrow O_1S_1A_1$ C_1 : *no opponents in range* C_2 : *at least one opponent in range* $C_3 = health < usableThreshold$ A_1 : *modify position* A_2 : *modify target position* A_3 : *modify position, health* A_4 : *modify target position, health*
$\mathbf{troop_1}$	$health = 100$ $range = 2$ $ammo = 20$ movingTime : exponential(3) shootingTime : exponential(2) $usableThreshold = 40$ $positionX = 40$ $positionY = 45$ $speed = 3$ $team = blue$ ... $transient(troop_1) : (ammo == 49)$	I_1, constraints: $class = PositionInfo$ $origin = Tank$ I_2, constraints: $class = InputFire$ $origin = Tank$	O_1, constraints: $class = PositionBroadcast$ $destination = Tank$ O_2, constraints: $class = OutputFire$ $destination = Tank$	$S_0 \rightarrow O_1S_1$ $I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ C_1 : *no opponents in range* C_2 : *at least one opponent in range* $C_3 = health < usableThreshold$ A_1 : *modify position* A_2 : *modify target position* A_3 : *modify position, health* A_4 : *modify target position, health*

Table 8.1: Meta-component Information in Tank vs SoldierTroop Scenario

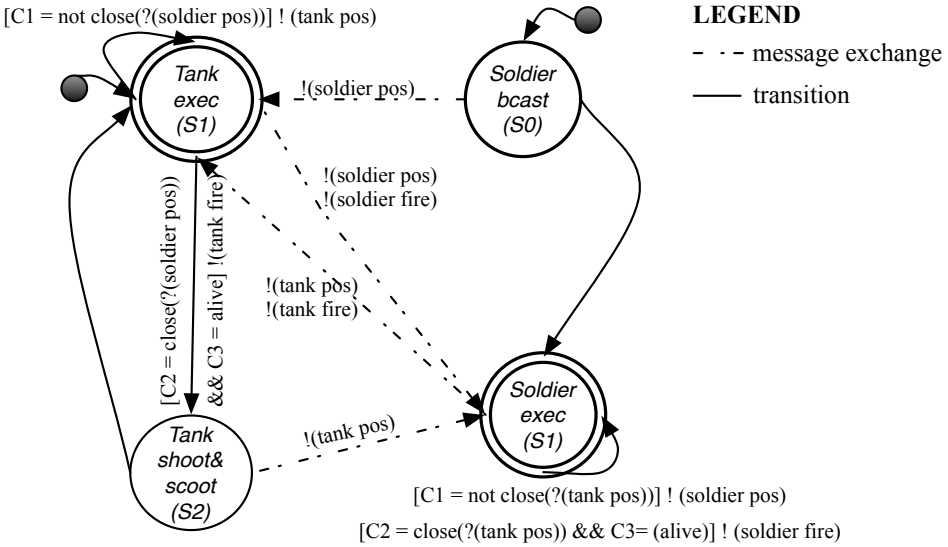


Figure 8.6: Data-driven Component Interaction

at by receiving an $InputFire$ message, it will be damaged (A_3 and A_4) depending on the closeness to the impact point. The tank component will move immediately from its position after firing at its opponent (state S_2). This is the implementation of a "shoot and scoot" tactic in which the tank moves after firing to prevent counter-artillery attacks [129]. For simplicity, the components assume that there are no obstacles on the two-dimensional grid battleground. Both tank and soldier troop can obtain the GPS coordinates at any time of their respective enemy.

We propose a new version of COML to cater for data-driven components in two ways [121]. Firstly, attribute names and values can be specified in the input and output data. Secondly, data oriented transition conditions and attribute changing sections can be specified in the behavior representation. For example, in the previous COML version, the transition conditions such as $C_1, \ldots, C_4$ from Table 8.1 could contain only simple logic such as the one from C_3. In the new COML version, conditions, such as $C_1, \ldots, C_4$, and attribute change sections, such as $A_1, \ldots, A_4$, can contain complex logic based on specific input data attribute values, such as the opponent's positions in Figure 8.7. The conditions and attribute-changing sections are parsed and evaluated during our validation process by condition and attribute parsers that determine the condition truth value and the new attribute values respectively. Consequently, the adjoining parsers have also been modified to include input and output data attribute values as well as more complex logic.

```
<component> ...
  <behavior> ...
    <condition name="C1"><value>
:methods
boolean all(int[][] positions, int n){
for (int i=0; i&lt;n;i++){
if (!( positions[i][0]&lt; positionX−range−1 ||
        positions[i][0]&gt; positionX+range+1 ||
      positions[i][1]&lt; positionY−range−1 ||
      positions[i][1]&gt; positionY+range+1))
        return false;}
        return true;}

:inputs
int[][] positions = new int[100][2];
positions = : init : array : input : I1 : position ;
int position_length = 1;
:preamble
boolean alive = (health &gt;= usableThreshold );
:main
System.out. println (all (positions, position_length )
&amp;&amp; alive);
</value>
</condition>

<data type="input" name="I1">
        <class>PositionInfo</class>
                <constraints> <constraint>
        <type>origin</type>
        <value>Soldier</value>
        </constraint></constraints>
<auxAttributes>
        <auxAttribute name="position">
        <X></X>
        <Y></Y>
</auxAttribute></auxAttributes>
</data>
```

Figure 8.7: Data-driven Component Representation

As discussed in Chapter 3, each component input and output message is defined in COML by data constraints. The constraints describe the primitive data present in the message (if any), as *data_type* and *range* constraints, as well as the *type* of components that can receive or send the output or input message respectively, as *destination* and *origin* constraints [125]. By *type* we mean either a base component type such as Tank or SoldierTroop, or a general *ModelComponent* type that describes reused model components. The base component types are specific to each application domain and are defined when the new domain is added to the framework. Condition C_1 in the $tank_1$ state machine aims to establish if any of the tank targets are within range. The condition parser that evaluates condition C_1 will construct a `.java` file with the structure determined by the `:methods`, `:inputs`, `:preamble`, `:main` tags. This file will be compiled and executed and the result of the execution (`true` or `false`) will determine the logical value (`true` or `false`) of condition C_1. Similar structure is found in the attribute values modification section in our COML schema.

8.4 Evaluation Methodology

This section evaluates the implemented prototype in two experiments. Firstly, we propose to evaluate the cost of semantic validation in data-driven simulation using the Military Training application domain as an example. We validate a composed model from the military training domain and evaluate the trade-off between accuracy and state space explosion by examining two scenarios, with and without data-driven modeling respectively. Next, we compare the runtime of the semantic validation of Queueing Network models with the runtime of semantic validation of Military Training composed models. Lastly, we evaluate the incremental cost of semantic composability validation for large models in the queueing networks application domain. The second experiment showcases the benefits and cost of model reuse by calculating the runtime of each life-cycle

step in the development of three composed Queueing Networks models with increasing number of components. All experiments have been executed on a machine with Intel Core 2 Duo CPU E6550 @ 2.33 GHz processor and 4 GB RAM, running Ubuntu Linux 8.04 (64 bit).

8.4.1 Cost of Semantic Validation

This experiment evaluates the cost of semantic validation of a composed data-driven simulation model as shown in Figure 8.5. We first evaluate an accuracy trade-off between a data-driven representation with a large state space, and a higher level representation, which does not consider data-driven executions, but has in turn a small state space. This trade-off is evident in the validation of general model properties, specifically in Concurrent Process Validation. Next, to show the overall cost of validation, we compare the runtime the formal validation of model execution for a data-driven model, with various non-data driven models from the Queueing Networks application domain. Lastly, we evaluate the cost of validating queueing network models with up to 1,000 components.

Let us assume that the composed model from Figure 8.5 passes syntactic verification and the best candidates returned by the discovery service for the Tank and SoldierTroop base components are components $tank_1$ and $troop_1$ respectively as shown in Table 8.1. The composed model can then be semantically validated following the process described in Chapters 6 and 7.

Accuracy vs State Space Explosion

As discussed in Chapter 6, the *Concurrent Process Validation* layer validates the component *coordination* of the composed model. This layer guarantees that safety, liveness, as well as deadlock freedom hold for all possible interleaved executions of *instanta-*

neous transitions of the composed simulator abstracted as a composition of concurrent processes. A composed model is invalid if it is found to be deadlocked, or if any of the components invalidate their safety or liveness properties. The behavior of each meta-component modeled as a state machine is translated into a logical specification using a logic converter module.

Figure 8.8(a) shows a possible translation of the component state machine into a Promela specification.

```
1 mtype {MSG}; chan to1 = [10] of {mtype}; ...
2 proctype CON_ONE_TO_ONE(chan in, out)
3 {do :: in ? MSG -> out ! MSG; od}
4
5 proctype TANK(byte id; chan in, out){
6 S1: atomic{ if :: in ? MSG ->
7  if :: out ! MSG -> progress: printf("MSG sent\n");
8                     goto S1; fi fi}}
9
10 proctype SOLDIERTR(byte id; chan in, out){
11 bit initial = 1;
12 S0: atomic{ if
13 :: (initial == 1) -> initial = 0;
14  if :: out ! MSG -> goto S1; fi fi}
15 S1: atomic{ if
16 :: in ? MSG ->
17 if :: out ! MSG -> progress: printf("MSG sent\n");
18                     goto S1; fi fi
19 }}
20 init{ run TANK(1, to1, from1);
21 run SOLDIERTR(2, to2, from2);
22 run CON_ONE_TO_ONE(from1, to2);
23 run CON_ONE_TO_ONE(from2,to1); }
```

(a) Simple Promela Specification

```
1 proctype SOLDIERTR(byte id, health, ..., posX, posY; chan in, out)
2
3 {bit initial = 1; byte posXFire, posYFire;
4 byte msgPosX, msgPosY, auxX, auxY, auxDistance,...;
5 S0: atomic{
6 if  :: (initial == 1) -> initial = 0;
7  if :: out ! MSG_POS -> goto S1; fi fi}
8 S1: atomic{ if atomic{if
9 :: in ? MSG_FIRE, msgPosX, msgPosY -> health = health - 10;
10
11  if :: health < health_threshold ->
12       if :: out ! MSG_DIE -> goto end; fi
13  :: else
14     if :: out!MSG_POS, posX, posY -> progress: printf("MSG sent\n");
15
16      goto S1;fi
17   fi
18 :: in ? MSG_DIE -> out ! MSG_DIE;goto end;
19 #GPS coord
20 :: in ? MSG_POS, msgPosX, msgPosY ->
21  if :: !(msgPosX<posX-range||msgPosX>posX+range ||msgPosX<posY-range
22       ||msgPosY<posY+range)->
23       if :: ammo>0->out!MSG_FIRE,msgPosX,msgPosY; ammo--; goto S1; fi
24  :: else -> auxDistance = distance;
25 :: msgPosX<posX->auxX = msgPosX+range;
26 :: else -> auxX = msgPosX - range; fi} ...
27 //similar to calc nxt position
28 if #broadcast position
29 :: out ! MSG_POS, posX, posY -> goto S1; fi
30 fi fi }
31 end: skip; }
32 init{
33 run TANK(1, 100, 20, 5, 40, 45,  to1, from1);
34 run SOLDIERTR(2, 100, 10, 5, 15, 20, to2, from2);
35 run CON_ONE_TO_ONE(1, from1, to2);
36 run CON_ONE_TO_ONE(2, from2,to1); }
```

(b) Detailed Promela Specification

Figure 8.8: Tank vs SoldierTroop in Promela

Each state is transformed into a Promela label, and the label includes input and/or output actions as specified by the meta-component behavior, as well as conditions on attribute values and attribute modifications. Transitions between states are instantaneous. Thus, time attributes such as $\Delta shootingTime$ and $\Delta movingTime$ from Table 8.1 are ignored. In the `init` method on line 20, communication channels are assigned to the

connectors and components according to their connection topology. Similar to the behavior of connectors in the real system, communication in the Promela specification is asynchronous. Liveness is specified using `progress` labels such as the one on line 7, and safety is specified using `assert` statements. Next, the Promela specification is validated by the SPIN model checker [14].

The above example considers a non data-driven approach to modeling, in which entities in Promela are modeled strictly by considering a high-level view of component coordination. This was the case previously for the Queueing Networks application domain, where the non data-driven state machines could be almost exactly transformed into Promela and the process was easily automated [119]. For example, if we were to interpret component coordination strictly from a message passing perspective, the resulting Promela specification would be that presented in Figure 8.8(a). This type of interpretation is easily automated and focuses only on component coordination. However, it lacks expressivity and any coordination logic. On the other hand, if we were to exactly transform the component state machines from their COML specification into Promela like in Figure 8.8(b) for the $troop_1$ component, we would obtain a more exact description of the attack but the translation process becomes difficult to automate.

We evaluate the two Promela models from Figure 8.8(a) and Figure 8.8(b) and present the results in Table 8.2. For a comprehensive evaluation we analyze ten scenarios. We first evaluate a training scenario where a single Tank component is fighting a single SoldierTroop component. We analyze this scenario from a simplified and data-driven perspective. Next, we evaluate a training scenario where five Tank components are put against five SoldierTroop components, in two situations, where pairs of Tank and SoldierTroop components fight in parallel, and where any Tank and any SoldierTroop component can attack each other. We analyze these scenarios from a simplified and data-driven perspective. We repeat these experiments for a composed simulator with

ten Tank and ten SoldierTroop components.

In our evaluation, we consider the number of components, the size of the SPIN state vector, which is a measure of how many variables are evaluated and ideally should be as small as possible [14], the number of states that are parsed, the memory cost and the execution time of the SPIN model checker in validating the respective Promela models. The results presented are an average of five runs.

No.	Model	#Components	State Vector Size (bytes)	# States	Memory (MB)	Execution Time (s)
1	Simplified Tank vs SoliderTroop	2	108	35	2.59	≤ 0.01
2	Simplified 5 Tank vs 5 SoldierTroop (one-to-one)	10	436	50,729	3.12	2.410
3	Simplified 5 Tank vs 5 SoldierTroop (dispersed)	10	1,024	3,759,596	261.05	243.67
4	Simplified 10 Tank vs 10 SoldierTroop (one-to-one)	20	760	3,317,524	283.29	384.00
5	Simplified 10 Tank vs 10 SoldierTroop (dispersed)	20	1,380	5,145,958	449.81	435.21
6	Data-driven Tank vs SoldierTroop	2	208	3,475	2.50	0.06
7	Data-driven 5 Tank vs 5 SoldierTroop (one-to-one)	10	960	1,927,277	126.99	72.63
8	Data-driven 5 Tank vs 5 SoldierTroop (dispersed)	10	1,628	1,820,875	106.94	91.12
9	Data-driven 10 Tank vs 10 SoldierTroop (one-to-one)	20	1,900	4,366,559	499.92	401.80
10	Data-driven 10 Tank vs 10 SoldierTroop (dispersed)	20	2,512	5,460,383	466.21	1,639.40

Table 8.2: Accuracy vs State Space Explosion in Semantic Validation

Two observations are evident from Table 8.2. Firstly, state space explosion is evident even when simplified, non data-driven modeling is employed. Depending on the scenario, this can result in execution times in the minutes range (for a one-to-one scenario), or in the tens of minutes range (for a dispersed scenario), for models with 20 components. The number of components directly influences state space explosion because an increase in the number of components implies an increase in the number processes whose interleaved execution must be analyzed, and in the number of variables that need to be analyzed. Additionally, the state space explosion is also influenced by the patterns of communication between the connected components, since this directly affects the number of possible combinations of interleaved executions that is analyzed by the SPIN model checker.

Table 8.2 shows that the state vector is almost double the size in the data-driven approach, compared with the simplified approach. Furthermore, the number of states parsed by the SPIN model checker is almost 100 times larger in the data-driven case. As

it can be seen, the data-driven approach fares much worse than the simplified approach for all measures considered. While these results are only for small models, the potential of state-space explosion for larger models is evident. Nonetheless, it is important to highlight here that the expressivity and accuracy gain in the data-driven approach is highly appealing and can justify the state space explosion cost, for which optimization methods such as symbolic execution [107] exist.

Runtime Evaluation

The cost of data-driven modeling is more evident in the formal validation of model execution. We compare in this section the runtime of the formal model execution layer for two application domains, namely, Queueing Networks and Military Training Simulations. The base components in Queueing Networks do not employ data-driven modeling, have a normal communication overhead, and a simplified logic. In contrast, in the Military Training application domain, base components employ data-driven modeling, have a high communication overhead, and a complex logic. For example, the server component in the Queueing Networks application domain is modeled as a simple service unit, which receives a job, services it for a time interval that is sampled from a specific distribution, and outputs the job on its output communication channel. In contrast, a Tank base component receives coordinate information from its opponents, and based on that information decides whether to shoot, move towards its target, etc. Similarly, the Tank component might move after firing according to a shoot and scoot strategy, or might remain in the same position.

Table 8.3 presents our results. We evaluate the runtime of the formal validation of model execution layer for a single-server queue and a grid system example in the Queueing Networks application domain. For both systems we consider cases where the composed model is both valid and invalid. The invalid models consider Source and Server components that process two classes of jobs. Similarly, for the Military Training

application domain we consider valid and invalid Tank vs SoldierTroop scenarios.

Model	# Components	# LTS States	Results	Execution Time (s)
Queueing Networks Application Domain				
Single-Server Queue	3	12	Valid	1.97
Single-Server Queue - 2 job classes	3	12	Invalid	4.72
Grid System	11	51	Valid	5.54
Grid System - 2 job classes	11	51	Invalid	8.29
Military Training Application Domain				
Tank vs Soldier Troop	2	21	Valid	6.20
Tank vs Soldier Troop	2	21	Invalid	22.30

Table 8.3: Runtime Evaluation of Formal Validation of Model Execution

As it can be seen, the cost of the formal validation of model execution scales well for the Queueing Networks application domain, with values of around two seconds for a composed model with three components, and less than nine seconds for a composed model with eleven components. However, in the case of the Military Training application domain, a composed model with two components has a runtime that is almost ten times higher than a model with a comparable number of components in the Queueing Network application domain. This is a direct result of part of the formal validation of model execution process, in which we calculate related composition states using the definition of V_ϵ. More specifically, in the calculation of V_ϵ we consider related states with respect to the related component attributes and their values. The COSMO ontology is queried to determine the relation between all pairs of component attributes. It follows that a large number of attributes will incur an increased number of ontology queries, which prove costly when the size of the ontology increases. This is the case in the Military Training application domain, where the average number of attributes per base component is 20, as opposed to the Queueing Networks application domain, which

has an average of 7.6 attributes per component.

We present a detailed analysis in Table 8.4. The execution time for the formal validation layer is clearly affected by the size and complexity of the components. The runtime increases with the number of attributes per component because in the calculation of V_ϵ *all* combinations of component attributes are considered when querying the COSMO ontology. Similarly, a larger number of state transition conditions translates into increased number of calls to the condition parsers in the evaluation of possible state transitions.

	Single Server Queue	**Tank vs Soldier Troop**
Data-driven	x	✓
# comp	3	2
average #states/comp	1.6	2
average #attributes/comp	7.6	20
average #delay time/comp	1	2
Runtime (s)		
Exact Match	3.0	6.2
$\mathbf{V}_\epsilon$	1.7	16.1

Table 8.4: Runtime Evaluation for Different Application Domains

To evaluate the scalability of our validation approach, we measure the runtime cost of semantic validation for queueing network models with ten, twenty, and a thousand components and present an average of ten runs in Table 8.5. For simplicity, the composed models represent serial connections of single-server components. Our evaluation shows that our validation process scales well even for models with a large number of components. However, it is important to note here that while the composed models have a large number of components, their complexity in terms of connections, the number of events, and the number of state and attribute changes is reduced. In particular, the composed model with $n = 1,000$ components had around $7,600$ attributes, and average of

Validation Step	Experimental Analysis - Runtime (s)		
	$n = 10$	$n = 20$	$n = 1,000$
1. Validation of General Model Properties			
A. Validation of Component Communication	< 0.1	0.1	0.5
B. Concurrent Process Validation	5.3	6.7	230.4
C. Meta-Simulation Validation	51.1	52.4	130.9
2. Formal Model Validation	6.8	10.3	97.6
Total	63.3	69.5	459.4

Table 8.5: Validation of Composed Models with Large Number of Components

1.6 states per component, and an arrival rate of 0.3 $evts/s$. The two labeled transition systems in formal model validation had a size of 5,994 nodes, for a value of $\tau = 3$. This is equivalent to 5,994 distinct composition states that had to be analyzed for closeness. For more complex models, such as grid systems or military training simulation applications, detailed also in the following sections, we have found that all validation steps scale well, with the exception of Concurrent Process Validation, which reaches state space explosion early. Moreover, Concurrent Process Validation shows a tendency towards state space explosion even for the simple serial connection in this example. This is the inherent limitation of model checking. However, several techniques such as symbolic execution [107] exist to contain this problem.

More importantly, Table 8.5 shows an increase in validity accuracy and inherently in model credibility, at the expense of an increase in computational cost. This is the inherent trade-off of our deny validity approach. However, our observation of real-life simulation problems leads us to believe that for a simulation problem, there are more invalid models than there are valid. Thus, our deny validity approach, through its eliminating stages, will discard invalid models without the need to go through the entire validation process. Nonetheless, if a model is not discarded, and thus the entire process

is executed, the credibility in its validity is highly increased.

8.4.2 Benefits and Cost of Model Reuse

This experiment evaluates the benefits and cost of model reuse. We employ the CoDES prototype to compose and validate the grid system from Figure 8.9, which contains two virtual organizations (VO) sharing a grid meta-scheduler job queue [42]. Each virtual organization consists of a local job scheduler and different types of computational resources. Assume the meta-scheduler accepts both CPU and I/O intensive jobs that can be serviced by VO_1, whereas VO_2 only services CPU intensive jobs.

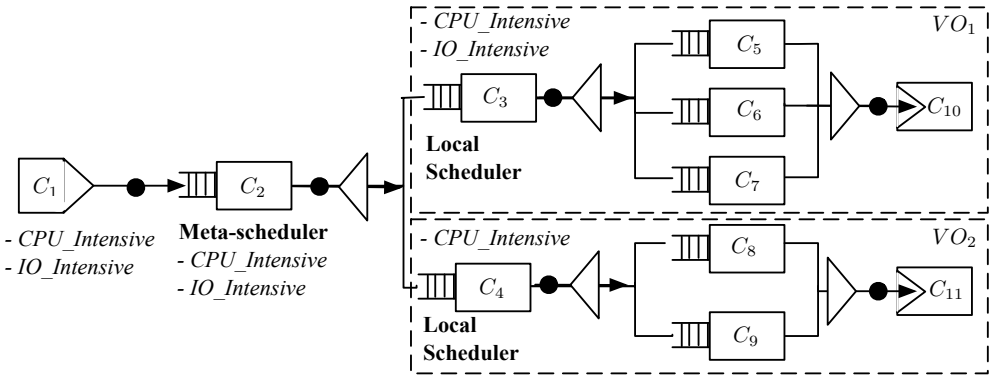
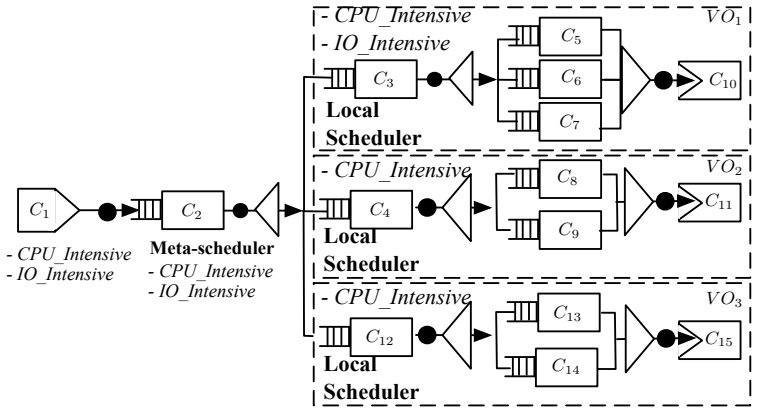


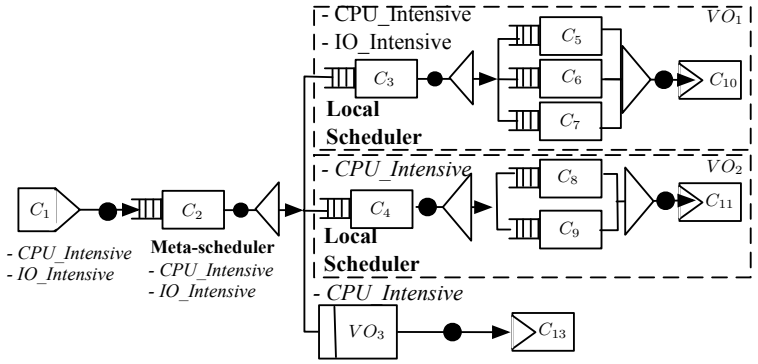
Figure 8.9: A Grid Computing System Composed using Base Components

In this experiment, we evaluate the variation of the runtime of each life-cycle step with the number of components. To showcase the benefits of model component reuse, we consider two scenarios, namely when only base components are used for model development, and when reused model components are employed. Firstly, we determine the runtime for each step for the grid system in Figure 8.9. Next, we evaluate a system in which another virtual organization with four components, VO_3, has been added as shown in Figure 8.10(a). We next add another virtual organization, VO_4, in the same manner. To determine if the reuse of model components brings any benefits with respect to runtime, instead of VO_3 composed from base components, we introduce a model

component that is structurally equivalent, as shown in Figure 8.10(b). We repeat the process for VO_4.



(a) Using Base Components



(b) Using Model Components

Figure 8.10: A Grid System with Three Virtual Organizations

We execute 10 sample runs for each life-cycle step and present the average runtime results in Table 8.6. As expected, the runtime of each step increases with the number of components. While the cost of runtime for most life-cycle steps is not high, ranging from several milliseconds to at most a minute, some steps can be very costly, with up to ten minutes in runtime. Through this experiment we are able to pinpoint the steps that need improvement and determine the particular conditions under which performance is decreased.

The benefits of reused model components are evident in syntactic composability verification. Here the reuse of model components will decrease the average runtime by

Life-cycle Step	**Experimental Analysis - Runtime (s)**				
	Base Components			Model Components	
	Current ($n = 11$)	Add VO_3 ($n = 15$)	Add VO_4 ($n = 19$)	Add VO_3 ($n = 13$)	Add VO_4 ($n = 15$)
Syntactic Composability Verification	$15.36{*}10^{-3}$	$15.86{*}10^{-3}$	$19.38{*}10^{-3}$	$13.08{*}10^{-3}$	$14.34{*}10^{-3}$
Model Discovery & Selection	0.95	1.31	1.65	69.19	137.43
Semantic Composability Validation					
1. Validation of General Model Properties					
A. Validation of Component Communication	0.09	0.11	0.12	0.10	0.11
B. Concurrent Process Validation	71.60	77.60	85.40	77.60	85.40
C. Meta-Simulation Validation	51.03	52.40	57.38	52.40	57.38
2. Formal Model Validation	6.76	10.32	14.70	10.32	14.70

Table 8.6: Cost of Composability for a Grid Computing System

up to 26%. It can be seen that the runtime when using model components for $n = 13$ is, unexpectedly, smaller than the runtime for $n = 11$ in the base component case. This is because the reuse of model components determines a smaller number of symbols in the production string to be verified by the grammar parser.

However, the runtime of the model discovery step shows a surprisingly large increase when reused model components are employed. However, on a closer look this is to be expected, because the number of model components in the repository is very high (about 2000 model components) and as such significantly more comparisons are performed in contrast to using only base components. Furthermore, in the current CoDES repository, model components are very similar (because of their black-box view), and as such our proposed initial elimination as discussed in Chapter 5, do not work. Efficient organization of the repository is beyond the scope of our current work.

Model component reuse offers yet no benefit for semantic composability. This is because in the current implementation, the validation of semantic composability is not a

closed operation, and as such validation results for model components cannot be reused. However, some parts of the validation process could be reused under specific conditions. For example, the time moments when the model component has produced output during meta-simulation can be saved as timestamps and reused later. As such, the meta-simulation validation of the model component can be reused as a black-box in the new validation process.

As it can be seen, the cost of the first stage of semantic composability validation has a very low cost. However, the credibility in the validity of the model is also reduced, because this stage only validates component communication. The credibility in the validity of the model increases with each validation stage, but also with incremental cost for each step.

8.5 Summary

The Java CoDES prototype follows closely the proposed life-cycle of a component-based simulation through three main modules, namely, *SyntaxVerifier* which checks the syntactic composability of the composed model, *ModelLocator* which discovers and selects the components based on the COSMO ontology, and *SemanticValidator*, which performs semantic validation of the composed model. All modules rely on the CoDES component representation in COML and the COSMO ontology. In addition, the SyntaxVerifier module employs an implementation of the Earley parsing algorithm, which parses composition rules from the CoDES composition grammars represented in EBNF. The COSMO ontology is queried using the Jena reasoner. In semantic composability validation, tools such as the SPIN model checker, the Choco constraint solver, and CADP toolset are employed. Our experiments employ two application domains, Queueing Networks and Military Training Simulations.

Our experiments show that while data-driven modeling in the Military Training sim-

ulation application domain provides a high level of detail that in turn increases the validation accuracy, it also incurs a runtime cost. Furthermore, the translation from a data-driven composed model to a representation of a similar level of detail for validation is a process that is difficult to automate. Nonetheless, we show that our validation process for application domains such as Queueing Networks is scalable, with the runtime cost of around seven minutes for models with up to 1,000 components on a commodity desktop PC. Moreover, our deny validity approach has increasing accuracy and credibility, but with incremental computational cost. While this is an inherent trade-off of our approach, most composed models will be discarded early in the validation process and will not incur the entire cost. On the other hand, composed models that will go through the entire validation process will have increased credibility.

Our second experiment shows the benefits and cost of model reuse. As expected, the use of model components results in a reduced cost for syntactic composability verification. However, model discovery and selection shows a drastic increase in discovery costs when reused model components are employed, showing the benefits of our modified life-cycle, in which syntactic composability verification is performed *before* model discovery and selection. It would be a waste of valuable time if model discovery is performed only to realize that the composed model is syntactically incorrect. Lastly, the overall cost of validation is small, even for a model with a high number of components.

Chapter 9

Conclusion and Future Work

9.1 Thesis Summary

The composability of simulation models is an appealing approach to reduce the time and cost of developing complex simulations. However, several challenges remain, including among others the lack of methodologies and techniques to support a component-based life-cycle, and the validation of semantic composability. In this thesis, we have designed and prototyped CoDES (Composable Discrete-Event scalable Simulation), an integrated approach for modeling and development of component-based simulations. Our focus has been on a new component-based simulation approach that facilitates the composition of simulation models at reduced costs, and on a semantic composability validation strategy to advance formal model validation.

The two major contributions of this thesis are: (i) an approach for composable simulations that addresses key crosscutting issues in the life-cycle of component-based simulation model development and (ii) a new *deny validity* semantic validation approach that advances our understanding of the trade-off between validation accuracy and computational cost. These contributions are detailed below.

9.1.1 Approach for Composable Simulations

While there is consensus on the benefits of an integrated component-based simulation approach, which include, among others, reduced development time and costs, to the best of our knowledge, this was the first attempt that looked at the entire component-based modeling and simulation life-cycle, from a conceptual model to a validated simulator ready for execution. We addressed key crosscutting issues of the component-based life-cycle and presented a feasible approach for syntactic composability and verification, model discovery and selection, and semantic composability and validation, in our proposed CoDES framework. Our contributions detailed below focus first on the overall contributions and trade-offs in the integrated design of CoDES, followed by our contributions in each individual life-cycle step.

1. Integrated vs. Piecewise

As we have argued in Chapters 1 and 2, a main challenge of component-based simulation model development is a life-cycle that considers important issues such as the component abstraction, the modeling of the conceptual model, the component discovery from a distributed repository, and the verification and validation of the composed model. A study of related work has shown that the ad-hoc piecewise integration of independent solutions to each step has reduced efficiency compared to an integrated approach. We proposed CoDES, a feasible and scalable framework for component-based modeling and simulation, in which the main design considerations have been solving the crosscutting issues, such as component sharing and reuse, component abstraction and representation, and knowledge representation, and the reduced cost of developing the composed simulation model. We proposed a four step life-cycle for component-based simulation model development, namely, conceptual model definition, syntactic composability verification, model discovery and selection, and semantic composability val-

idation [125]. Although the order of these steps was different from current approaches, we showed that our solution has a decreased runtime cost on average. Our solutions for the above crosscutting issues facilitated modeling, composability, and validation.

Key to our proposed approach was the *meta-component* abstraction that specified the component in terms of attributes and behavior. Towards flexibility in composition, we proposed a *black-box component-connector abstraction*, in which the component implementation and structure are hidden from the framework and the simulation model composer, and components are connected using only well-defined connectors. The black-box component abstraction had the advantage of reducing complexity in reasoning about composability in all life-cycle steps for models composed from base components. However, this resulted in a negative impact on the life-cycle of model components, which had an increased execution time in discovery and selection. This issue is detailed below. We proposed the COML standard, implemented in XML, to describe simulation components and the data they exchange. While this ensured that all components were described using the same syntax, an inherent drawback we observed was that XML could not capture semantic information about components. Accordingly, the meta-component abstraction was enhanced using semantically sugared attributes described in COSMO, our proposed component-based ontology [118, 125]. COSMO facilitated meaningful discovery and selection and accurate semantic validation.

We proposed the *sharing of simulation components* within and across application domains for the *development of more complex, larger simulation models*. We organized simulation components as *base components*, which are fundamental entities specific to each application domain, and *model components*, which are composed from base and other model components. Using this organization, we achieved the sharing of components across application domains both in breadth and in depth within an application domain, and at the same time allowed for the composition and reuse of hierarchical

heterogeneous components. Furthermore, the separation into application domains facilitated the scalable representation of knowledge specific to each application domain.

2. Life-cycle of Simulation Model Development

In addressing the crosscutting issues presented above, we promoted a life-cycle of component-based simulation with reduced cost. Our contributions to each step are detailed below. For ease of reading, we present our contributions to semantic composability validation in Section 9.1.2.

a. **Scalable Syntactic Composability Verification of a Conceptual Model**

In conceptual model definition, a main challenge has been to facilitate the definition of the simulation problem by the model composer. We proposed the drawing of a conceptual model using icons of base and model components, which are dragged by the model composer on a drawing panel and subsequently connected using well-defined connectors. Because base components for each application domain were well-defined using the COSMO ontology, the conceptual model captured intrinsic knowledge about the simulation problem, without explicit input from the model composer. This modeling paradigm coupled with our proposed component organization into base and model components, improved the conceptual modeling stage and provided valuable information, which was used to facilitate syntactic composability verification and the reduction of state space in model discovery and selection.

We proposed a novel approach to the verification of syntactic composability using application domain specific composition grammars [118]. The structure of the conceptual model was translated into a production string that was subsequently verified against the composition grammar. Our study showed that the advantages of using a composition grammar to describe connection rules within and across application do-

mains include, among others, formalization and scalability. Formalization increases credibility in the verification process and facilitates automation. On the other hand, infinitely large models can be expressed as production strings and verified against the composition grammar, at minimal costs. For example, the syntactic verification of a composed model with 10,000 components took less than seven seconds on a commodity desktop PC.

Moreover, we proposed the verification of syntactic composability *before* model discovery and selection. This reduced the average cost of development because syntactically incorrect conceptual models were discarded before the costly discovery and selection process. Our approach is different from current life-cycles [29] that perform syntactic composability verification *after* the composed model has been discovered.

b. **Relevant Model Discovery and Selection**

In model discovery and selection, two main challenges were identified. Firstly, similarity is not an "yes/no" answer and current work looks only at matching components in terms of their syntax in therms of method names and parameters. Secondly, the runtime cost of model discovery and selection increases with the size of the component repository and the size of the component-based ontology used for reasoning.

(i) **Partial Matches**

To address the first issue, we defined *partial matches* between query and repository components, with semantic relevancy that facilitated meaningful discovery. We defined a *Matching Index* to quantify semantic similarity, with values ranging from zero (no match) to one (perfect match). The calculation of the Matching Index quantified component similarity according to attribute relationships defined in the COSMO ontology [125], with the help of our semantically-sugared meta-component. This is in contrast to current approaches, which

either do not consider partial matches [24], or fail to look at the component semantics [29].

(ii) **Reduction of Search Space**

To address the second issue, we proposed an elimination approach, in which repository components were only considered if their type (for base components) and their neighbors (for model components) were the same as those defined in the conceptual model. This technique was very effective for base components. However, for model components, the combination of the black-box component abstraction, which is restrictive in providing information about the inner structure of the model component, and the performance of the reasoner used to query the COSMO ontology, proved detrimental to the execution cost of the discovery of model components. This result could be improved if a trade-off between a black-box and a white-box component abstraction is employed, as discussed in Section 9.2.1. For example, if the production string that defines the structure of the model component is known, it could be hashed as a key in a DHT overlay for the discovery of models with similar structure. Nonetheless, the use of the conceptual model in our proposed optimization, without explicit input from the model composer, is another example of the benefits of a seamlessly integrated approach.

3. Integrated Approach for Component-based Simulation Development

We have implemented our proposed framework using the Java 1.6 programming language. We have structured the CoDES sourcecode in a modular manner to ensure limited costs in maintenance. To ensure the feasibility of our approach, we employed a suite of public-domain software tools including Protege 4.2 and Jena 2.5.4 for ontology reasoning, and SPIN 5.1.7 (working with Promela 5.1.7) and CADP toolset 2006-a with BISIMULATOR 1.5 for semantic composability validation. Our experiments were

executed on a commodity desktop PC, with Intel Core 2 Duo CPU E6550 @ 2.33 GHz processor and 4 GB RAM, running Ubuntu Linux 8.04 (64 bit).

As an example, we discussed the composition of a grid computing system in a Queueing Networks application domain, and a military training scenario, in a Military Training Simulation application domain, and evaluated the cost of component-based model development using both theoretical and experimental analyses. Our results have shown the feasibility of our proposed approach. The cost of syntactic verification is in the range of milliseconds for a composed model with twenty components, and in the range of seconds for composed models with 10,000 components. The computational cost was influenced by the composition grammar and was found to be at worst $O(n^3)$, where n was the number of components.

Next, discovery and selection of base components is less than two seconds. In contrast, the discovery and selection of model components averaged around one minute for a repository of 2,000 model components. The overall performance of model discovery and selection was influenced by the size and organization of the repository, the component abstraction, the size of the ontology, the efficiency of the reasoner, issues that are beyond the scope of this thesis. Nonetheless, our results showed the efficiency of our approach which verifies the conceptual model *before* discovery. Lastly, semantic validation was in the range of three minutes for models with only base components. More significantly, the incremental cost of validation was reduced with increased accuracy.

For models in the Queueing Networks application domain, the total life-cycle cost ranged from two minutes for models with 20 components, to seven minutes for models with 1,000 components, on the commodity desktop PC described above. The size of the problem validated in this experiment was around 7,200 attributes and around 6,000 distinct composition states. The validation process is responsible for the largest amount percentage of this cost, with around one and six minutes respectively.

9.1.2 Deny Validity Approach for Semantic Validation

The validation of the semantic composability of the composed artifact is of paramount importance to increase the credibility of the composed model. We have shown that the semantic validation of the composed model is a hard problem, with solutions that require the presence of a system expert, or approaches to automation at the cost of accuracy [88] or scalability [77, 127]. Validation using a system expert is not feasible in the context of a component-based framework, where there exists a geographical separation between the model composer and the component developers and systems experts. Thus, we proposed a fully automated approach for the validation of semantic composability of the composed simulation model with increased accuracy to provide for high model credibility [119, 122]. Our approach delivered incremental accuracy at each stage with varying trade-offs related to the complexity of the composed model, in terms of the number of attributes and states [121].

Our key strategy in semantic composability validation was based on the observation that, the computational cost of checking for an invalid model is lower on average than the cost of checking for a valid model. As such, we proposed a *deny validity* strategy that discarded invalid models using a two-step process. Our approach first attempted to eliminate models in the *Validation of General Model Properties* step. This step had several stages, which tested the composed model for a variety of model properties. At the end of these stages, confidence in the composed model validity was increased. However, the model could still be invalid when compared to a reference model. We proposed *Formal Validation of Model Execution* to increase credibility in the validity of the composed model using our proposed time-based formalism.

1. Validation of General Model Properties

The first layer of our deny validity approach discarded invalid models through the val-

idation of general model properties, such as safety and liveness for instantaneous and timed transitions [119, 121]. We proposed three validation stages, namely, the validation of component communication, concurrent process validation to validate model properties for instantaneous transitions, and meta-simulation for timed transitions. To increase model credibility, we consider formal and practical definitions of model properties.

a. **Validation of Component Communication**

   While syntactic composability verification ensures that components in the composed model are properly connected, their communication might still be incompatible. We proposed the *Composability Index* to measure the degree of data alignment of neighboring components in the composition [125]. The calculation of the Composability Index relied on the representation of the component behavior, which considered also the data that a component can exchange. This data was described using COSMO, our proposed ontology, by semantically-enriched data constraints. The Composability Index provides a low-cost, initial measure of the suitability of the components to the composition. However, the current data constraints consider primitive data types and taxonomies of general classes of data. While this is a major improvement compared to current work [24, 29], the feasibility of more detailed description of exchanged data remains to be studied.

b. **Validation of All Interleaved Executions**

   Concurrent process validation proposed to validate all possible interleaved component executions for deadlock, safety, and liveness. Because this would normally lead to a high computation cost, we reduced the problem size by considering *timeless* transitions in this step, and added time in the next step, Meta-simulation validation. We proposed the use of model checking to validate all possible interleaved executions of the components in the composed model. This had the advantage of validating

all possible combinations of execution states[1]. However, the inherent drawback of using model checking to formally validate general model properties is the state space explosion problem [20]. While this problem was easily contained for application domains such as Queueing Networks where components have a simpler structure, it deprecated for data-driven domains such as Military Training simulations [121]. We have shown that state space explosion could still be contained for data-driven application domains at the cost of the level of detail employed in the component abstraction in the model checker specification. Moreover, several techniques such as symbolic execution [107] could be employed to further ameliorate the problem.

c. **Meta-Simulation Validation**

In *meta-simulation validation*, we proposed the definition of safety and liveness properties from a simulation user perspective. We included validity points provided by the model composer, and transient predicates present in the COML component representation. From a real world perspective, we considered time attributes and all other component attributes, with values specified by the model composer. The drawback of this method was that it was based on sampling, which guaranteed validity only as long as the same sample parameters and streams are used in the simulator execution.

2. Formal Validation of Model Execution

The second layer of our deny validity approach proposed formal validation of model execution to increase model credibility by comparing the composed model to a reference model using our proposed formalism. Key challenges in formal validation included establishing a feasible formalism that preserved the level of detail in the composed model, a definition of validity that captured different levels of validity, and the nature of

[1]For models with simple structure in the Queueing Networks application domains, this step also had low execution cost compared with the following steps, regardless of the number of components.

the reference model in the context where system experts and other usual providers are not present.

a. **A New Time-based Formalism**

To address the first issue, our novel time-based formalism represented dynamic component behavior as a function of states and time. Contrary to current work that lose accuracy and cannot represent complex behavior [88], our proposed formalism allowed for the representation of dynamic component behavior. Furthermore, complex composition structures, with "fork" and "join" topologies, could be validated using our formalism. The overhead of our proposed formalism was minimal because the translation from the COML component abstraction to the formalism was straightforward. Furthermore, there was no loss in the level of detail (and implicit accuracy) because the functional formalism retains the notion of state and time. However, similar to meta-simulation validation, our approach was based on sampling to obtain the time moment values when the component received/produced output.

b. **Definitions and Measures of Validity**

We proposed a formal definition of validity in which a valid composed model is one whose execution is close enough to the execution of a reference model. The inherent advantage of this definition was that it allowed for various degrees of validity, similar to situations in real life. However, an important problem was how to meaningfully quantify closeness to a reference model. We proposed the semantic metric relation V_ϵ, which quantified composition state similarities based on semantically sugared components defined in our component-based ontology. Informally, a composed model is valid if there are enough composition states in its execution that follow in the same order as related composition states in the execution of the reference model. The semantically-sugared meta-component abstraction facilitated the calculation of related composition states using the COSMO ontology.

We have fully implemented the validation process and have tested our approach on a variety of models, which differed both in terms of size and complexity. Our observations have shown that two important factors influenced the accuracy of our definitions and metrics, namely, the reference model, and the threshold after which composition states are considered related.

(i) **Reference Model**

Several approaches in traditional simulation, which does not adhere to a component-based worldview, have proposed the comparison of the composed model with some form of valid reference entity [10, 104]. This entity is assumed to exist a-priori, or is provided by the system expert or model developer during validation. This assumption is not feasible in the context of component-based simulation model development where a system expert is missing when the composed model is created. We proposed to construct the reference model from perfect, desired, and generic descriptions of base components that were provided by system experts when each application domain is added to the CoDES framework. This was a feasible assumption, since system experts that provide the composition grammars and the extension of the COSMO ontology, will also have knowledge about generic base components in the new application domain. The generic base components are state-based generalized descriptions of fundamental entities in the application domain and do not have attached implementations. As such, they cannot be used as simulation components but can be employed as reference components. To demonstrate the feasibility of our proposed approach, we validated models of various sizes and complexity from application domains such as Queueing Networks and Military Training Simulations.

(ii) **Similarity Threshold**

In our formal approach, a valid model was defined as one that is *close enough*

with respect to the states, sequence and duration of component execution, to a reference model. Yet, what exactly is close enough (i.e., the values of ϵ), as with all thresholds, remains an open problem. In our experiments, we have set the value of ϵ using a trial and error approach for each application domain. Our sensitivity studies showed that the values of ϵ depend on the reference models in each application domain, and on the hidden assumptions inherent in the reference components. The first factor offers hope because a threshold could be proposed for each application domain. However, the second factor suggests that a more pragmatical approach could be employed, namely by allowing the model composer to change the values of ϵ as required. Nonetheless, future studies are needed to establish the feasibility of this approach.

9.2 Future Directions

Directions for future work include: increasing model reusability and scalability through hybrid white-box and black-box abstractions of model components, techniques for representing and validating models with emergent properties, and practical deployment issues on emerging platforms such as cloud computing.

9.2.1 Increasing Model Reusability and Scalability

While most component-based environments focus on reusing previously developed base components, they do not address the reuse of the developed composition, which we called model components [119, 122, 125]. As outlined before, current component marketplaces show marginal increases in the number of components, in the range of 20% over ten years. However, the number of available components can be significantly increased through the reuse of model components.

While the reuse of model components promises to scale the component repository

and increase the variety of available models, several challenges remain. A key challenge is the abstraction of model components. This is because a black-box abstraction, in which no information about the structure of the model component is saved, is not sufficient for efficient model discovery and selection and for accurate semantic validation. Furthermore, the issue of representing the model component dynamic behavior over time becomes more stringent in the context of model components with a large number of sub-components. These issues are usually ignored by software engineering component-based solutions and Service Oriented Architectures [111] because of the inherent complexity of composition and validation. From this perspective, the study of simulation model components, where time and states are of paramount importance, promises to reveal and solve fundamental issues in composability.

We have shown in Chapter 8 that the cost of discovery of reused model components from a large repository is high, and is influenced both by the size of the repository and by the approach to the search space reduction. This is a limitation of our approach, which stems from the purely black-box view that we employ for a model component. Using our black-box model component abstraction, the search space reduction is insignificant in the discovery of model components. In contrast, a white-box abstraction exposes all details of the model component structure, as well as attributes and behavior of its underlying sub-components, at the cost of a high level of detail that is harder to manage. A model component abstraction that achieves a reasonable trade-off between the two will facilitate the reduction of the search space in model discovery and thus decrease discovery time. Furthermore, it will provide the access key towards highly scalable models. Moreover, a suitable level of model component abstraction also influences the cost of semantic validation of models of practical size.

Lastly, semantic composability is not a closed operation [88], meaning that two semantically valid models will not form a semantically valid composition. With respect

to model components, this implies that previously validated model components do not always compose into valid models. Thus, semantic composability validation must be performed on the newly composed model every time. However, a right trade-off level between black-box and white-box abstractions for a model component could possibly improve this aspect. This could be the case for the validation of component communication and meta-simulation validation, among others. For example, the validation of the component communication may not be performed for the sub-components of the model components, but only by looking at the input/output of the model component. In meta-simulation validation, the time moments when the model component produces or needs output or input respectively, could be reused in the validation process.

9.2.2 Composed Models with Emergent Properties

A key issue in composability is that component interactions over time can lead to emergent properties (or behaviors), which are not obvious from the individual component behaviors. These emergent properties, also called behavioral composability, are dynamic and are often revealed when the composition is used in decision-making, where a valid answer is crucial [49, 101]. Emergent properties are known to be pervasive in many everyday systems and are of major research interest in domains such as computational biology [7] and systems design [101]. An emergent property can be defined as "a property of an assemblage that could not be predicted by examining the components individually" [101]. An analogy would be that of a musical concert. While the range and pitch of each instrument are factors that depend entirely on the manufacturing specifications of the instrument and the skill of the individual musician, when played together in a concert, the final result is not easily predictable from the individual instruments. Rather, it depends on the conductor interpretation of the score and the combined harmony of the ensemble. The validation of composed models with emergent properties

remains a hard problem [49].

The identification, modeling, and validation of emergent properties facilitate the discovery of valid composed models and provides valuable user insight in the behavior of these new models. In this respect, we hypothesize that the current practice of validation by comparison with a reference model [10] is insufficient for models with emergent properties. A key idea is to focus on (i) what a user can expect a composed model to achieve using composition objectives, and (ii) to develop formal software techniques to identify and validate behavioral composability.

Current approaches represent how a component acts in a composition using attributes and behavior [77, 125, 132]. However, for behavioral composability validation, this abstraction is not suitable for identifying emergent behavior. A suitable component abstraction that specifies components and model components with focus on the validation of emergent properties is needed. An approach could be to describe a component in terms of "what" it achieves using component properties. This provides an effective mean for validation because the emergent behavior might be exposed when the aggregated properties of the composition are compared with composition objectives. Furthermore, agent-based simulation, in which entities are modeled as individuals with local knowledge whose interaction results in emergent properties [73], might provide future insight into the problem of emergence.

Bibliography

[1] M. Aksit. Separation and Composition of Concerns in the Object-oriented Model. In *ACM Computing Surveys*, volume 28, pages 148–159, 1996.

[2] M. Aleksy, A. Korthaus, and M. Schader. *Implementing Distributed Systems with Java and CORBA*. Springer, 2005.

[3] R. Allen. *A Formal Approach to Software Architecture*. PhD Thesis, School of Computer Science, Carnegie Mellon University, 1997.

[4] M. Armbrust, A. Fox, R. Grifth, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, UC Berkley Reliable Adaptive Distributed Systems Laboratory, 2009.

[5] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The CCA Component Model for High-Performance Scientific Computing. *Concurrency and Computation: Practice Experience*, 18:215–229, 2006.

[6] J. Aronson and P. Bose. A Model-Based Approach to Simulation Composition. In *Proceedings of the Symposium on Software Reusability*, pages 73–82, Los Angeles, USA, 1999.

[7] M. A. Aziz-Alaoui and C. Bertelle. From System Complexity to Emergent Properties. In *Proceedgins of the* 2^{nd} *International Workshop on Multi-Agent Systems for Medicine and Computational Biology*, pages 104–116, Hakodate, Japan, 2006.

[8] F. Baader and W. Nutt. *The Description Logics Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge, UK, 2003.

[9] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *Communications of the ACM*, 46:43–48, 2003.

[10] O. Balci. Verification, Validation and Accreditation of Simulation Models. In *Proceedings of the Winter Simulation Conference*, pages 135–141, Atlanta, USA, 1997.

[11] J. Banks, J. Carson, B. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice Hall, USA, 2005.

[12] 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.

[13] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. The Computational Complexity of Component Selection in Simulation Reuse. In *Proceedings of the Winter Simulation Conference*, pages 2472–2481, Orlando, USA, 2005.

[14] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer Verlag, 2008.

[15] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, pages 28–37, 2001.

[16] D. Bosnacki and D. Dams. Integrating Real Time into Spin: A Prototype Implementation. In *International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, pages 432–438, Paris, France, 1998.

[17] K. Breitman, M. Casanova, and W. Truszkowski. *Semantic Web: Concepts, Technologies and Applications*. Springer Verlag, London, UK, 2007.

[18] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of the* 7^{th} *International Workshop of Component-Oriented Programming (WCOP02)*, Malaga, Spain, 2002.

[19] M. Buchi and W. Weck. A Plea for Grey-box Components. Technical report, Turku Centre for Computer Science, 1997.

[20] J. R. Burch, E. M. Clarke, K. L. McMillanD, L. Dill, and L. J. Hwangs. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98:142–170, 1992.

[21] G. Chang, M. J. Healey, J. A. M. McHugh, and J. T. L. Wang. *Mining the World Wide Web - An Information Search Approach*. Kluwer Academic Publishers, 2001.

[22] 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.

[23] Choco Constraint Programming System. http://sourceforge.net/projects/choco/, (retrieved Oct. 2008).

[24] R. Chreyh and G. Wainer. CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames, 2009.

[25] A.-C. Christiaan, C. Paredis, and P. Khosla. A Composable Simulation Environment For Mechatronic Systems. In *Proceedings of the SCS European Simulation Symposium*, Darmstadt, Germany, 1999.

[26] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[27] ComponentSource - The Definitive Source of Software Components. www.componentsource.com, (retrieved Apr. 2010).

[28] J. Dahmann, R. Fujimoto, and R. Weatherly. The Department of Defense High Level Architecture. In *Proceedings of the Winter Simulation Conference*, pages 142–149, Atlanta, USA, 1997.

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

[30] P. Davis and R. Anderson. Improving the Composability of Department of Defense Models and Simulations. Technical report, RAND Corporation, 2003.

[31] P. Davis, P. Fishwick, C. Overstreet, and C. Pedgen. Model Composability as a Research Investment: Responses to the Featured Paper. In *Proceedings of the Winter Simulation Conference*, pages 1585–1591, Orlando, USA, 2000.

[32] P. Davis and A. Tolk. Observations on New Developments in Composability and Multi-Resolution Modelling. In *Proceedings of the Winter Simulation Conference*, pages 859–870, Washington, USA, 2007.

[33] 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.

[34] D. A. DePalma, S. Dolberg, M. Mavretic, and J. Johnson. *Objects on the Net. Software Strategy Report*. Forrester Research, Inc., Cambridge, USA, 1996.

[35] Department of the Navy. *Modeling and Simulation Verification, Validation, and Accreditation Implementation Handbook*. 2004.

[36] E. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66, New York, USA, 1982.

[37] J. Earley. An Efficient Context-free Parsing Algorithm. In *Communications of the Association of Computing Machinery*, volume 13, pages 94–102, 1970.

[38] M. Eklf, J. Ulriksson, and F. Moradi. NetSim: an Environment for Network Based Modeling and Simulation. Technical report, Symposium on C3I and MS Interoperability (MSG-22), NATO Modeling and Simulation Group, 2003.

[39] M. C. Fisher. Aggregate Level Simulation Protocol (ALSP) Managing Confederation Development. In *Proceedings of the Winter Simulation Conference*, pages 775–780, Orlando, USA, 1994.

[40] Flashline - Component Manager. www.flashline.com, (retrieved Apr. 2010).

[41] C. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.

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

[43] M. Fox, D. Brogan, and P. Reynolds. Approximating Component Selection. In *Proceedings of the Winter Simulation Conference*, pages 428–434, Orlando, USA, 2004.

[44] W. Frakes and K. Kang. Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering*, 31:529–536, 2005.

[45] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 158–163, Berlin, Germany, 2007.

[46] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's Hard to Build Systems Out of Existing Parts. In *Proceedings of the* 17^{th} *International Conference on Software Engineering*, pages 179–185, Seattle, USA, 1995.

[47] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer-Verlag, New York, USA, 2004.

[48] S. Glotzer, S. Kim, P. Cummings, A. Deshmukh, M. Head-Gordon, G. Karniadakis, L. Petzold, C. Sagui, and M. Shinozuka. WTEC Panel on International Assesment of Research and Development in Simulation-based Engineering and Science, 2009.

[49] R. Gore and P. Reynolds. Applying Causal Inference to Understand Emergent Behavior. In *Proceedings of the Winter Simulation Conference*, pages 712–721, Miami, USA, 2008.

[50] 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.

[51] D. S. Hartley. Verification & Validation in Military Simulations. In *Proceedings of the Winter Simulation Conference*, pages 925–931, Georgia, USA, 1997.

[52] J. Himmelspach, M. Röhl, and A. M. Uhrmacher. Component-based Models and Simulation Experiments for Multi-agent Systems in James II. In *Proceedings of the International Workshop From Agent Theory to Agent Implementation*, Estoril, Portugal, 2008.

[53] G. Holzmann. *Design And Validation Of Computer Protocols*. Prentice Hall, 1991.

[54] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2001.

[55] IEEE Standard 1278.1. IEEE Standard for Distributed Interactive Simulation - Application Protocols.

[56] Institute for Software Research Irvine CA - Architecture Description Languages. http://www.isr.uci.edu/architecture/adls.html.

[57] ISO Standard for Extended BNF. http://www.iso.org/iso/en/ cataloguedetail-page.cataloguedetail? csnumber=26153, 1996.

[58] S. Jarzabek. Pragmatic Strategies for Variability Management in Product Lines in Small- to Medium-size Companies. In *Software Product Lines*, San Francisco, USA, 2009.

[59] Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net, 2002.

[60] P. Kanellakis and S. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[61] S. Kasputis and H. Ng. Composable Simulations. In *Proceedings of the Winter Simulation Conference*, pages 1577–1584, Orlando, USA, 2000.

[62] W. D. Kelton, R. P. Sadowski, and D. T. Sturrock. *Simulation with Arena*. McGraw-Hill Higher Education, 2003.

[63] J. Kofron. Checking Software Components Behavior Using Behavior Protocols and Spin. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1513–1517, Korea, 2007.

[64] V. V. Krzhizhanovskaya, P. M. A. Sloot, and Y. E. Gorbachev. Grid-Based Simulation of Industrial Thin-Film Production. In *Simulation*, volume 81, pages 77–85, 2005.

[65] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal Tools for Technology Transfer*, 1:134–152, 1997.

[66] D. N. Le, B. D. Tran, P. Tan, A. E. Goh, and E. W. Lee. MODiCo: A Multi-Ontology Web Service Discovery and Composition System. In *Proceedings of the* 9^{th} *International Conference on Web Engineering*, pages 531–534, San Sebastian, Spain, 2009.

[67] A. Lehman. Component-Based Modeling and Simulation Status and Perspectives. In *Proceedings of the* 8^{th} *IEEE Distributed Simulation and Real-time Applications*, pages 15–15, 2004.

[68] J. Magee and J. Krmaer. Dynamic Structure in Software Architecture. In *Proceedings of the* 4^{th} *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 3–14, San Francisco, USA, 1996.

[69] I. Mahmood, R. Ayani, V. Vlassov, and F. Moradi. Statemachine Matching in BOM Based Model Composition. In *Proceedings of the* 13^{th} *IEEE Symposium on Distributed Simulation and Real-Time Applications*, pages 136–143, Singapore, 2009.

[70] C. McClure. *Software Reuse : A Standards-based Guide*. IEEE Computer Society Press, 2001.

[71] 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.

[72] F. Min, P. Ma, and M. Yang. A Knwoledge-based Method for the Validation of Military Simulation. In *Proceedings of the Winter Simulation Conference*, pages 1395–1402, Washingthon, USA, 2007.

[73] R. Minson and G. K. Theodoropoulos. Distributing RePast Agent-based Simulations with HLA. *Concurrency and Computation: Practice and Experience*, 20:1225–1256, 2008.

[74] ModSAF - Modular Semi-Automated Forces. http://www.onesaf.org/onesaf.html, (retrieved Apr. 2010).

[75] F. Moradi, R. Ayani, S. Mokarizadeh, G. H. A. Shahmirzadi, and G. Tan. A Rule-based Approach to Syntactic and Semantic Composition of BOMs. In *Proceedings of the* 11^{th} *IEEE Symposium on Distributed Simulation and Real-Time Applications*, pages 145–155, Crete, Greece, 2007.

[76] F. Moradi, R. Ayani, G. Tan, and S. Mokarizadeh. A Rule-based Semantic Matching of Base Object Models. *International Journal of Simulation and Process Modelling (IJSPM)*, 5:132–145, 2009.

[77] F. Moradi, P. Nordvaller, and R. Ayani. Simulation Model Composition Using BOMs. In *Proceedings of the* 10^{th} *IEEE International symposium on Distributed Simulation and Real-Time Applications*, pages 242–252, Washington, USA, 2006.

[78] M. Morisio, M. Erzan, and C. Tully. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering*, 28:340–357, 2002.

[79] A. Natrajan, P. F. Reynolds, and S. Srinivasan. MRE: A Flexible Approach to Multi-resolution Modeling. *ACM SIGSIM Simulation Digest*, 27:156–163, 1997.

[80] S. Onggo, D. Soopramanien, and M. Pidd. A Dynamic Business Model For Component-based Simulation Software. In *Proceedings of the Winter Simulation Conference*, pages 954–959, 2006.

[81] N. Oses, M. Pidd, and R. J. Brooks. Critical Issues in the Development of Component-based Discrete Simulation. In *Simulation Modelling Practice and Theory*, pages 495–514, 2004.

[82] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:455–495, 1982.

[83] OWL Web Ontology Language Overview. http://www.w3.org/tr/owl-features/, 2004.

[84] E. Page, R. Briggs, and J. A. Truffarolo. Toward a Family of Maturity Models for the Simulation Interconnection Problem. In *Proceedgins of the Spring Interoperability Workshop*, 2004.

[85] E. Page and J. Opper. Observations on the Complexity of Composable Simulations. In *Proceedings of the Winter Simulation Conference*, pages 553–560, Phoenix, USA, 1999.

[86] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the* 5^{th} *GI-Conference on Theoretical Computer Science*, pages 167–183, Karlsruhe, Germany, 1981.

[87] M. Petty and E. Weisel. Basis for a Theory of Semantic Composability. In *Proceedings of the Spring Simulation Interoperability Workshop*, Orlando, USA, 2003.

[88] M. Petty and E. W. Weisel. A Composability Lexicon. In *Proceedings of the Spring Simulation Interoperability Workshop*, pages 181–187, Orlando, USA, 2003.

[89] M. Pidd. Simulation Software and Model Reuse: A Polemic. In *Proceedings of the Winter Simulation Conference*, pages 772–775, Washinghton, USA, 2004.

[90] F. Plasil, D. Balek, and R. Janecek. OFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 43–51, Annapolis, USA, 1998.

[91] R. H. Pollack, R. Baldwin, J. R. Neyer, and D. Perme. Requirements for Composing Simulations: A Use-Case Approach. In *Proceedings of the Spring Simulation Interoperability Workshop*, 2003.

[92] Protege Ontology Editor. http://protege.stanford.edu, 2004.

[93] C. Reade. *Elements of Functional Programming*. Boston, MA, USA: Addison-Wesley, Boston, USA, 1989.

[94] R. Reese and D. L. Wyatt. Software Reuse and Simulation. In *Proceedings of the Winter Simulation Conference*, pages 185–192, Atlanta, USA, 1987.

[95] H. Rheingold. *Virtual Reality*. Simon Schuster, New York, USA, 1992.

[96] S. Robinson. Conceptual Modeling for Simulation Part I: definition and requirements. *Journal of the Operational Research Society*, 59:278–290, 2007.

[97] D. Rogerson. *Inside COM (Microsoft Programming Series)*. Microsoft Press, 1997.

[98] M. Röhl and A. M. Uhrmacher. Composing Simulations from XML-specified Model Components. In *Proceedings of the Winter Simulation Conference*, pages 1083–1090, Monterey, USA, 2006.

[99] S. Christley and X. Xiang and G. Madey. An Ontology for Agent-Based Modeling and Simulation. In *Proceedings of the Agent Conference on Social Dynamics: Interaction, Reflexivity and Emergence*, Chicago, USA, 2004.

[100] M. Saaltink. The Z/EVES System. *Lecture Notes in Computer Science*, 1212:72–85, 1997.

[101] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer Systems Design*. Morgan Kaufman, 2009.

[102] A. Samantaray, K. Medjaher, B. O. Bouamama, M. Staroswiecki, and G. Dauphin-Tanguy. Component-Based Modelling of Thermofluid Systems for Sensor Placement and Fault Detection. In *Simulation*, volume 80, pages 381–398, 2004.

[103] P. G. Sarang, K. Gabhart, A. Tost, and T. McAllister. *Proffesional EJB*. Wrox Press, 2001.

[104] R. P. Sargent. Verification and Validation of Simulation Models. In *Proceedgins of the Winter Simulation Conference*, pages 130–143, Orlando, USA, 2005.

[105] H. S. Sarjoughian. Model Composability. In *Proceedings of the 2006 Winter Simulation Conference*, pages 149–158, Monterey, USA, 2006.

[106] 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.

[107] S. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 157–168, Portland, USA, 2006.

[108] G. Silver, K. R. Bellipady, J. Miller, K. J. Kochut, and W. York. Supporting Interoperability using the Discrete-event Modeling Ontology (DeMO). In *Proceedings of the Winter Simulation Conference*, pages 1399–1410, Austin, USA, 2009.

[109] 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*, pages 1108–1117, Washington, USA, 2007.

[110] 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.

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

[112] J. Sowa. *Knowledge Representation: Logical, Philosophical and Computational Foundations*. Brooks/Cole, Pacific Grove, USA, 1999.

[113] M. Spiegel, P. R. Jr., and D. Brogan. A Case Study of Model Context for Simulation Composability and Reusability. In *Proceedings of the Winter Simulation Conference*, pages 437–444, Orlando, USA, 2005.

[114] J. Srba. On the Power of Labels in Transition Systems. In *Proceedings of the* 12^{th} *International Conference on Concurrency Theory*, pages 277–291, Aalborg, Denmark, 2001.

[115] B. Srivastava and J. Koehler. Web Service Composition - Current Solutions and Open Problems. In *IEEE Internet Computing*, volume 8, pages 51–59, 2004.

[116] V. Sugumaran and V. C. Storey. The CCA Core Specification in a Distributed Memory SPMD Framework. *Advances in Information Systems*, 34:8–24, 2003.

[117] K. J. Sullivan and J. C. Knight. Experience Assessing and Architectural Approach to Large-scale Systematic Reuse. In *Proceedings of the* 18^{th} *International Conference on Software Engineering*, pages 220–229, Berlin, Germany, 1996.

[118] C. Szabo and Y. M. Teo. On Syntactic Composability and Model Reuse. In *Proceedings of the International Conference on Modeling and Simulation*, pages 230–237, Phuket, Thailand, 2007 (invited paper).

[119] C. Szabo and Y. M. Teo. An Approach for Validation of Semantic Composability in Simulation Models. In *Proceedings of the* 23^{rd} *ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 3–10, New York, USA, 2009.

[120] C. Szabo and Y. M. Teo. An Approach to Component-based Simulation Model Development. *ACM Transactions on Modeling and Simulation*, page (submitted for review), 2010.

[121] C. Szabo and Y. M. Teo. On Validation of Semantic Composability in Data-driven Simulations. In *Proceedings of the* 24^{th} *ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 73–80, Atlanta, USA, 2010.

[122] C. Szabo, Y. M. Teo, and S. See. A Time-based Formalism for the Validation of Semantic Composability. In *Proceedings of the Winter Simulation Conference*, pages 1411–1422, Austin, USA, 2009.

[123] C. Szypeski. *Component Software: Beyond Object-Oriented Programming (*2^{nd} *Edition)*. Addison-Wesley, 2002.

[124] N. Tansalarak and K. Claypool. Finding a Needle in the Haystack: A Technique for Ranking Matches Between Components. In *Proceedings of the Component-based Software Engineering Conference*, pages 171–186, Ottawa, Canada, 2005.

[125] Y. M. Teo and C. Szabo. CODES: An Integrated Approach to Composable Modeling and Simulation. In *Proceedings of the* 41^{st} *Annual Simulation Symposium*, pages 103–110, Ottawa, Canada, 2008.

[126] A. Tolk. What Comes After the Semantic Web - PADS Implications for the Dynamic Web. In *Proceedings of the* 20^{th} *Workshop on Principles of Advanced and Distributed Simulation*, pages 55–62, Singapore, 2006.

[127] M. K. Traore. Analyzing Static and Temporal Properties of Simulation Models. In *Proceedings of the Winter Simulation Conference*, pages 897–904, Monterey, USA, 2006.

[128] M. K. Traore and B. Zeigler. Conditions for Model Component Reuse. In *Daghstul Seminar on Component-based Modeling and Simulation*, 2004.

[129] US Army Field Manual No. 3-09.22, Headquarters, Department of the Army. Tactics, Techniques, and Procedures for Corps Artillery, Division Artillery, and Field Artillery Brigade Operations, 2001.

[130] US Department of Defense - VV&A Recommended Practices Guide. http://vva.msco.mil, (retrieved Aug. 2009).

[131] R. L. Wittman and C. T. Harrison. OneSAF: A Product Line Approach to Simulation Development. Technical report, The MITRE Corporation, 2001.

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

[133] B. Zeigler and S. Sarjoughian. Introduction to DEVS Modeling and Simulation with JAVA: Developing Component-based Simulation Models. Technical report, Arizona Center of Integrative Modeling and Simulation, University of Arizona, 2005.

[134] B. P. Zeigler, Y. K. Moon, and G. D. Ball. The devs environment for high-performance modeling and simulation. *IEEE Computational Science and Engineering*, 1994.

Appendix A

Meta-Component Representation

Figure A.1 presents a skeleton of the COML schema for a CODES meta-component. In the CoDES framework, a component has mandatory attributes, defined by the tag $mandatoryAttributes$ on line 4, component specific attributes, defined by the tag *specificAttributes* on line 3, and behavior, defined by the tag $behavior$ on line 7.

```
1  <element name="component">
2  <complexType>
3          <sequence>
4                  <element name="mandatoryAttributes" type="spec:mandAtt"/>
5                  <element name="specificAttributes" type="spec:specAtt"/>
6                  <element name="data" type="spec:dataType" maxOccurs="unbounded"/>
7                  <element name="behavior" type="spec:behaviorType"/>
8          </sequence>
9  </complexType>
10 </element>
```

Figure A.1: Component Representation in COML

The COML component representation relies on a COML Schema (implemented in XSD), which defines syntax rules that must be met by all COML component files. Figure A.2 presents excerpts from the COML Schema for base components. The COML Schema file contains rules and restrictions for each particular tag. Figure A.3 presents a snapshot of the asserted COSMO class structure in Protégé.

```
<element name="component">
 <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 A.2: Component Representation in COML

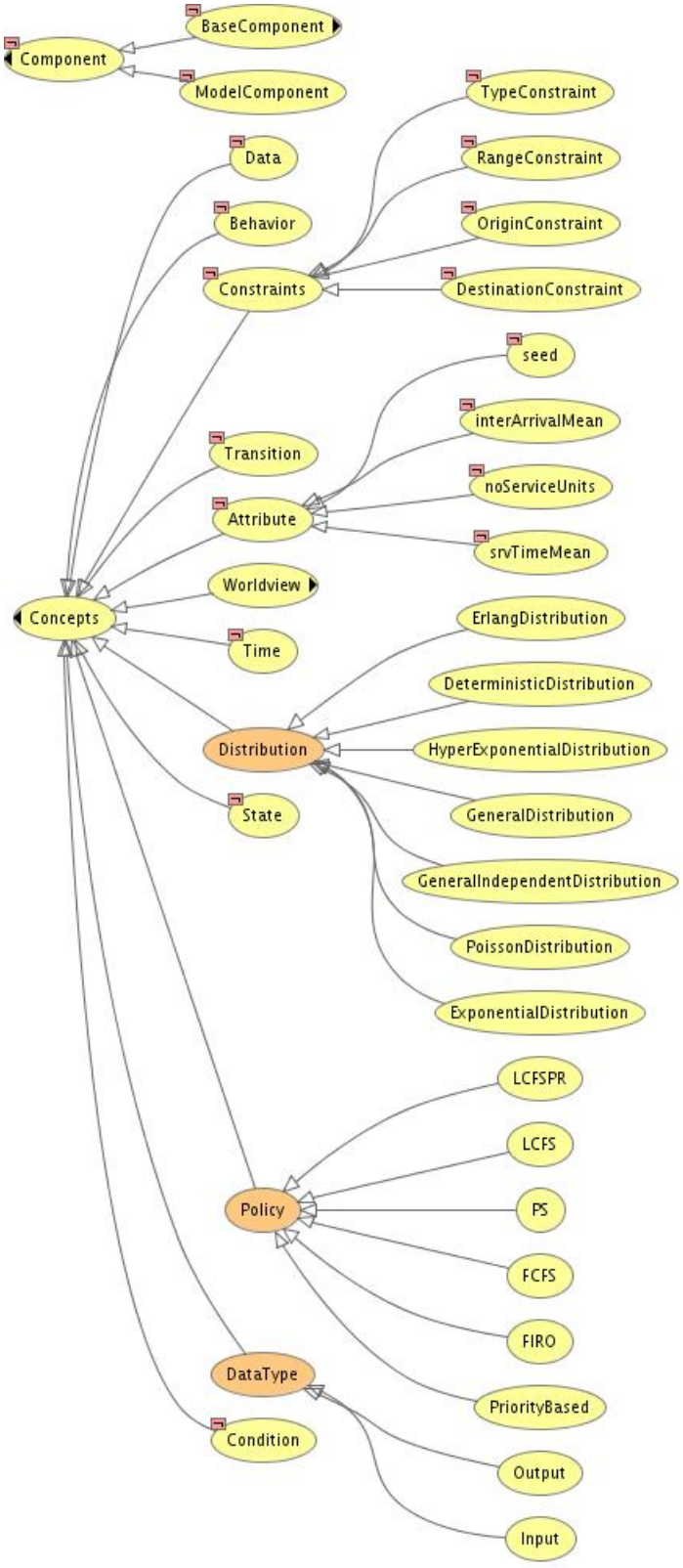


Figure A.3: COSMO Asserted Ontology Class Structure

Appendix B

Component-based Modeling of a Single-Server Queue System

B.1 Conceptual Model Definition

The simulation developer inputs the conceptual model by drag-and-drop icons of Queueing Networks base components on the GUI drawing panel and subsequently connecting them using One-to-One connectors as shown in Figure 4.7.

B.2 Syntactic Composability Verification

The conceptual model presented above is syntactically verified by the *Syntax Verifier* module. First, the module checks that no component port is left unconnected. Next, a production string is generated to describe the conceptual model. The production string is a linear arrangement of the components' types according to their position on the graphical screen. The production string for this model is:

$$Simple\_QN = Source\ ConO\ Server\ ConO\ Sink$$

The model is syntactically verified if the production string is accepted by the Queueing Networks composition grammar. In the CoDES implementation, the composition grammar is first parsed using our implemented Earley parser. Next, the production string is verified by the parser. The parser accepts the production string and thus the model is syntactically correct.

B.3 Model Discovery and Selection

Table B.1 presents the query information provided by the user for component C_1, compared to the component information for the discovered repository component with the highest matching index. Using the weight values $w_r = 0.33$, $w_a = 0.33$, and $w_b = 0.33$, and considering that no specific attributes are mentioned in the query, the matching index value becomes:

$$MI(C_1, C) = 0.5 * 1 + 0.0 * 0 + 0.5 * 0.875 = 0.937.$$

An initial eliminatory search is performed, eliminating base components whose type does not match the $Source$ type, which is derived from the conceptual model. Matching on the required attributes returns a matching index of $MI_r = 1$, since the type attribute matches exactly and all relevant words in the description of the query component are found in the repository component. Since no component attributes are provided, the resulting weight values are $w_r = 0.5$, $w_a = 0$, and $w_b = 0.5$, and $MI_a = 0$. There are no input constraints, hence the term 0. For the output constraints, the *destination* constraint matches exactly, and the *range = 21:24* of the repository component is included in the query range *range =* $10 : 35$, hence 0.75, with a total behavior matching index $MI_b = 0.875$.

Table B.2 presents the query information provided by the user for component C_2,

Query Component (C_1)	**Repository Component** (C)	**Matching Index**
type: Source **description:** open source	**type:** Source **description:** open source	$MI_r = 1$
-	-	$MI_a = 0$
$IC = \emptyset, OC = \{C_1\}$ $C_1 = \{$ *destination = Server, range = 10:35* $\}$	$IC = \emptyset, OC = \{C_1\}$ $C_1 = \{$ *type = int, range =21:24, destination = Server*$\}$	$MI_b = \frac{0+(1+0.75)}{0+2} = 0.875$
		$\mathbf{MI(C_1, C) = 0.937}$

Table B.1: Query Information for Component C_1

compared to the component information for the discovered repository component with the highest matching index. Using the weight values $w_r = 0.33, w_a = 0.33$, and $w_b = 0.33$, and considering that no specific attributes are mentioned in the query, the matching index value becomes:

$$MI(C_2, C) = 0.5 * 1 + 0.0 * 0 + 0.5 * 0.75 = 0.875.$$

An initial eliminatory search is performed, eliminating base components whose type does not match the $Server$ type, which is derived from the conceptual model. Matching on the required attributes returns a matching index of $MI_r = 1$, since the type attribute matches exactly and all relevant words in the description of the query component are found in the repository component. Since no component attributes are provided, the resulting weight values are $w_r = 0.5, w_a = 0$, and $w_b = 0.5$, and $MI_a = 0$. For the first input constraint C_1, the *type* constraint of *double* matches the *type = double* constraint of the repository component, and the *origin* constraint is also matched, hence the term $(1 + 1)$. For the output constraints, the *destination* constraint matches exactly, and the *range = 21:24* is not included in *range =* $14 : 18$, hence 1, with a total behavior matching index $MI_b = 0.75$.

Table B.3 presents the query information provided by the user for component C_3, compared to the component information for the discovered repository component with the highest matching index. Using the weight values $w_r = 0.33, w_a = 0.33$, and $w_b = 0.33$, and considering that no specific attributes are mentioned in the query, the matching

Query Component (C_2)	**Repository Component** (C)	**Matching Index**
type: Server **description:** server with single unit	**type:** Server **description:** single uint server	$MI_r = 1$
-	-	$MI_a = 0$
$IC = \{C_1\}, OC = \{C_2\}$ $C_1 =$ { *type = double, origin = Source—Server* } $C_2 =$ { *destination = Sink, range = 21:24* }	$IC = \{C_1\}, OC = \{C_2\}$ $C_1 =$ { *type = double, range = 10:35, origin = Source|Server*} $C_2 =$ { *type = double, range =10:35, destination = Server|Sink*}	$MI_b = \frac{(1+1)+1}{2+2} = 0.75$
		$\mathbf{MI(C_2, C) = 0.875}$

Table B.2: Query Information for Component C_2

index value becomes:

$$MI(C_1, C) = 0.5 * 1 + 0.0 * 0 + 0.5 * 0.5 = 0.75.$$

An initial eliminatory search is performed, eliminating base components whose type does not match the $Sink$ type, which is derived from the conceptual model. Matching on the required attributes returns a matching index of $MI_r = 1$, since the type attribute matches exactly and all relevant words in the description of the query component are found in the repository component. Since no component attributes are provided, the resulting weight values are $w_r = 0.5, w_a = 0$, and $w_b = 0.5$, and $MI_a = 0$. For the input constraints, the *origin* constraint matches exactly, and the *range = 21:24* of the query component is not present in the repository component, hence 0. There are no output constraints, hence the term 0 with a total behavior matching index $MI_b = 0.875$.

Query Component (C_1)	**Repository Component** (C)	**Matching Index**
type: Sink **description:** sink component	**type:** Sink **description:**sink	$MI_r = 1$
-	-	$MI_a = 0$
$IC = \{C_1\}, OC = \{\emptyset\}$ $C_1 =$ { *origin = Server, range = 21:24* }	$IC = \{C_1\}, OC = \emptyset$ $C_1 =$ { *origin=Server*}	$MI_b = \frac{(1+0)+0}{0+2} = 0.5$
		$\mathbf{MI(C_3, C) = 0.75}$

Table B.3: Query Information for Component C_3

Table B.4 presents the relevant meta-component information of the discovered and selected repository components C_1, C_2, and C_3.

	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_3}$
Attribute	$noJobsGenerated = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ interArrivalTime: exponential(3) $transient(C_1) : (noJobsGenerated == 1)$	$noJobsServiced = 0$ $timeout = 20$ $time = 200$ $timeScale = 1$ serviceTime : exponential(6) $busy = false$ $transient(C_2) : (busy == true)$	$noJobsPrinted = 0$ $timeout = 20$ $time = 200$ $timeScale = 1/5$ $\Delta printingTime = 1$ $transient(C_3) : (noJobsPrinted == 1)$
Input	-	I_1, constraints: $origin = Source\|Server$ $range = 10; 35$ $type = double$	I_1, constraints: $origin = Server$
Output	O_1, constraints: $destination = Server$ $range = 11; 15$ $type = int$	O_1, constraints: $destination = Server\|Sink$ $range = 10; 20$ $type = double$	-
State Machine	$S_1(\Delta interArrivalTime) \rightarrow S_2$ $S_2 \rightarrow S_1O_1[A_1]$	$I_1S_1 \rightarrow S_2[A_1; A_3]$ $S_2(\Delta serviceTime) \rightarrow S_1O_1[A_2]$	$I_1S_1 \rightarrow S_2$ $S_2(\Delta printingTime) \rightarrow S_1[A_1]$
	$[A_1] = noJobsGenerated++;$	$[A_1] = (busy = true);$ $[A_2] = (busy = false);$ $[A_3] = noJobsServiced++;$	$[A_1] = noJobsPrinted++;$

Table B.4: Meta-component Information of Discovered Components

B.4 Semantic Composability Validation

B.4.1 Validation of General Model Properties

For the model presented in Table B.4, the composability index CI is obviously 1, since the constraint I_1 on the input of C_2 subsums the constraint O_1 on the output of C_1, and similarly the constraint I_1 on the input of C_3 contains the constraint on the output O_1 of component C_2.

Concurrent Process Validation

Figure B.1 shows the state machine of every component translated into a Promela specification. Each state is transformed into a Promela label, and the label includes input and/or output actions as specified by the meta-component behavior, as well as conditions on attribute values and attribute modifications. Transitions between states are assumed to be instantaneous. Nonetheless, for component C_1 described in process $SOURCE1$ on line 7 we simulate time through the additional process $SourceCounter$ shown on line 11. The role of the $SourceCounter$ process is to modify the inter-arrival

time ($interArrivalTime$) until it reaches a predefined value. When this happens, process $SOURCE1$ is activated and produces a message on its out channel. Counter processes are introduced for all components that have only output channels.

Each type of connector is defined as a Promela process. For example, process *CON_ONE_TO_ONE* on line 4 describes the one-to-one connector. The fork and join connectors are not part of this composition and as such are omitted. In the `init` method on line 24, communication channels are assigned to the connectors and components according to their connection topology. Similar to the behavior of connectors in the real system, communication in our Promela specification is asynchronous. However, the maximum number of messages in a channel is bounded by a constant value. This is because unbounded queues are not permitted in the Spin model checker [14], since the focus is on process *coordination* and not computation.

The Promela specification is validated by the Spin model checker. Valid executions can be specified in various ways. By default the Spin model checker validates that there is no deadlock or any unreachable states in the system. We limit safety validation to deadlock checking. To specify liveness, we assign a `progress` label to each state in a component that produces output. The Spin model checker validates the system by analyzing all possible process states obtained through the interleaved execution of the active processes. The absence of deadlocks and non-progress cycles is validated. It is important to note the high number of visited states even for this simple example. State space explosion decreases the feasibility of employing this type of validation as a standalone validation process, and thus we include it only as the first layer in our approach.

Meta-simulation Validation

Meta-simulation validation shows that the logical properties demonstrated in the previ-

```
1  hidden byte sourceIAMax = 10; byte sourceIATime; byte noJobsSource = 0;

3  proctype CON_ONE_TO_ONE(chan in, out){
4  do :: in ? Job -> out ! Job; od}

6  proctype SOURCE1(int id, noJobsMax; chan out){
7  do :: (sourceIATime == sourceIAMax) -> sourceIATime =0;
8    if     :: out ! Job -> progress: printf("[Source] Job sent\n"); fi od }

10 proctype SourceCounter(){
11 do :: (sourceIATime < sourceIAMax) -> sourceIATime++; od}

13 proctype SINK1(int id; chan in){
14 S1: atomic{
15 if :: in ? Job -> printf("[Sink] Job received!\n"); goto S1; fi}}

17 proctype SERVER3(int id; chan in, out){
18 bit busy;
19 S1: {
20 if :: in ? Job -> printf("[Server] Job received!\n"); busy=1;goto S2; fi}
21 S2: {
22 if :: out ! Job -> progress: printf("[Server] Job sent! \n"); busy=0; goto S1;}}
23 init{
24 run SourceCounter();
25 run SINK1(3, to3);
26 run SERVER3(3, to2, from2);
27 run SOURCE1(1, from1);
28 run CON_ONE_TO_ONE(from1, to2);
29 run CON_ONE_TO_ONE(from2, to3);}
```

Figure B.1: Single-Server Queue Model in Promela

ous layer hold through time. Our implementation translates the complete state machine of each component into a Java class hierarchy. Attributes and their values provided by the user, state transitions and time are modelled. Next, we construct a meta-simulation of the composed model using the translated classes. During the meta-simulation run, sampling is performed for attributes that require so. This is the case especially for time attributes such as inter-arrival time or service time. For example, as shown in Table B.4, the inter-arrival time $\Delta interArrivalTime$ for component C_1 is sampled from an exponential distribution with a $mean$ of 3. The distribution type and mean values are an example of attribute values provided by the user. Since sampling is performed, the meta-simulation is run for $N = n * noSampling$ times, where n is the total number of

components and $noSampling$ is the total number of locations where sampling is done. If any of the properties does not hold in the meta-simulation runs, the composition is declared invalid.

The most important logical properties that are validated through time are safety and liveness. From a practical perspective, we consider safety to mean that components do not produce invalid output. The simulator developer specifies the desired valid output by providing *validity points* at various connection points in the composition. A validity point contains semantic description of data that must pass through its assigned connection point. For example, the two validity points for the data that passes through the second connector in Figure 4.7 could be $VP_1 = d_1\{origin = Server, destination = Sink, range = 10; 35, type = double\}$, and $VP_2 = d_2\{origin = Server, destination = Sink, range = 1; 2\}$. If anytime during the meta-simulation run semantically incompatible data passes through the connection point, a safety error is issued. Semantically incompatible data is data whose type and constraints are not related in the COSMO ontology.

Liveness is validated by considering a *transient* predicate assigned to each component. The value of the transient predicate is ideally provided by the component creator in the meta-component as shown in Table 6.1. Its initial value is `false`. Each component is assigned a *liveness observer* that is notified every time the attributes involved in a transition change values. The liveness observer evaluates the transient predicate and time stamps the moment in which the transient predicate becomes true. A component is considered *alive* if it's liveness observer has evaluated the transient predicate to *true* and then to *false* in an interval of time smaller than the specified timeout. For example the transient predicate for component C_2 could be $transient(C_2) = (busy == true)$.

Based on the meta-component information from Table B.4, the state machine for each component is executed on separate threads. Figure B.2 presents one of the meta-

simulation runs for our example model. The flow of input and output from component

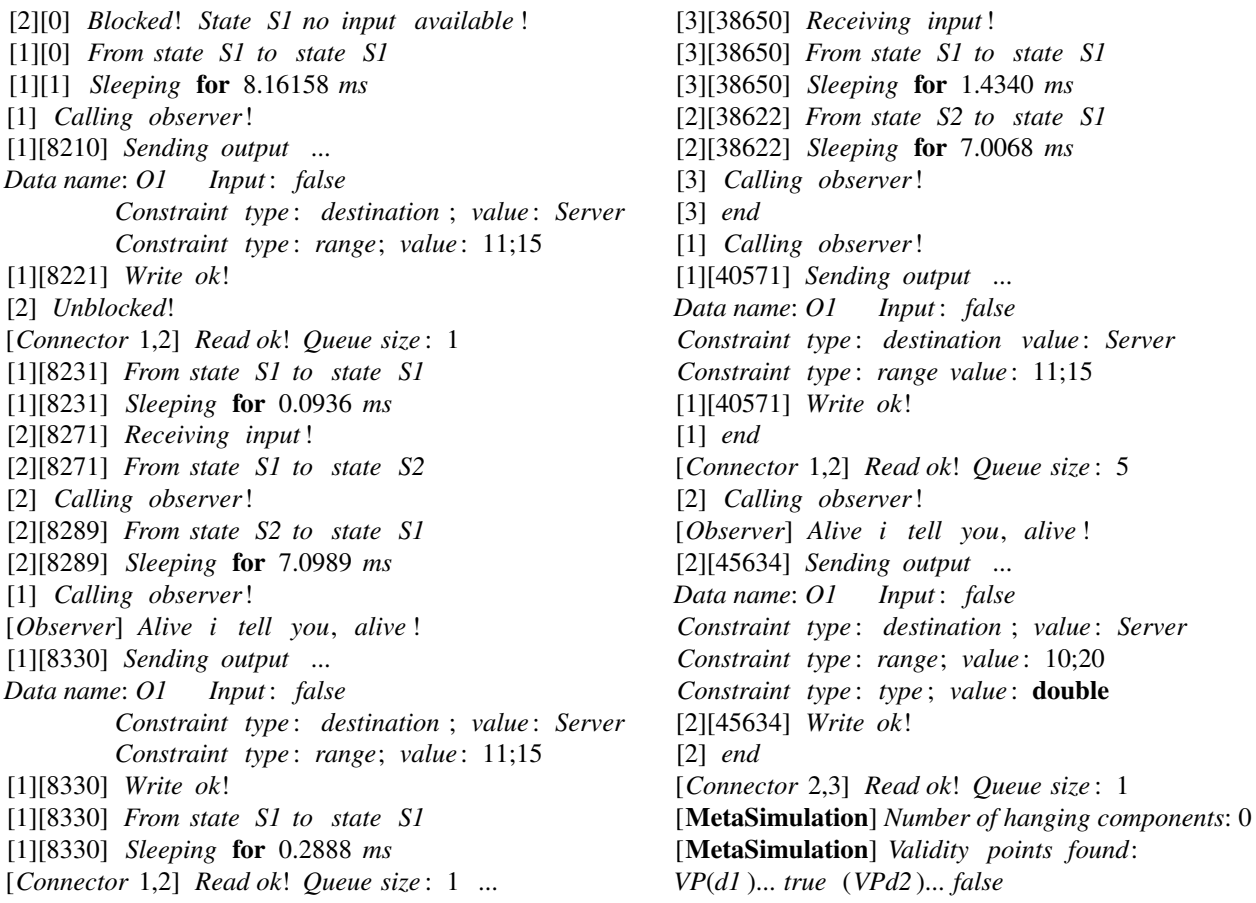


Figure B.2: Meta-Simulation Output

to connector and reverse ([$Connector$]) as well as the execution of the safety and liveness ([$Observer$]) observer are shown.

B.4.2 Formal Validation of Model Execution

In the following we present the detailed validation process only for the selected components C_i represented formally as functions f_i. The same process is repeated for reference functions f_i^*. For this example, we consider the behavior of the reference components represented by f_i^* to be the same with respect to input/output transformations

to the behavior represented by f_i. The base components C_i differ from the reference components C_i^* through additional logging attributes such as $noJobsGenerated$.

Formal Component Representation

In the *Formal Component Representation* step, the state machine for component C_1 as specified in its meta-component is $S_1(\Delta interArrivalTime) \rightarrow S_2, S_2 \rightarrow S_1O_1[A_1]$. This expression is translated to a formal component representation specified by f_1 which is defined as

$$f_1 : \emptyset \times S_1 \times T_1 \rightarrow \{O_1\} \times S_1 \times T_1, f_1(\emptyset, s_i, t) \rightarrow (O_1, s_i', t + \Delta t)$$

where Δt is sampled from a specified distribution and the function is re-called until $t > T$, where the simulation runs for time $T = 40$ wall clock units.

Unfolding and Sampling

The above expression is not useful for the *Unfolding and Sampling* step in our approach because during a simulation run t and Δt have specific values. Thus we unfold the function call graph for $\tau = 3$ times and sample the values for Δt, using mean values provided by the user. For component C_1 assume that the inter-arrival time is sampled from an exponential distribution with a $mean = 3$. With sampling and an unfolding grade of $\tau = 3$ we have $\Delta t = 6, \Delta t = 2, \Delta t = 4$. For component C_2 described formally by f_2 assume the service time has an exponential distribution with a mean of $mean = 6$ sampled as $11, 6, 1$. Lastly, we assume component C_3 formalized in f_3 takes 1 unit of time to service each job, so $\Delta t = 1$ for all samples. The values of f_1, f_2, and f_3 are presented in Table B.5.

Composition

Next, the function composability is validated in the *Composition* step. Following Def-

	Unfold	Δt	Formula	Meaning
f_1	1	6	$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$	from state s_1^1 with no input at time **0**, to state s_2^1 with output O_1 at time **6**
	2	2	$f_1(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$	from state s_2^1 with no input at time **6**, to state s_3^1 with output O_1 at time **8**
	3	4	$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$	from state s_1^1 with no input at time **8**, to state s_2^1 with output O_1 at time **12**
f_2	1	11	$f_2(I_2, s_1^2, x \geq 0) \rightarrow (O_2, s_2^2, x + 11)$	from state s_1^2 with input I_2 at time $\mathbf{x}$, to state s_2^2 with output O_2 at time $\mathbf{x} + \mathbf{11}$
	2	6	$f_2(I_2, s_2^2, t \geq x + 11) \rightarrow (O_2, s_3^2, t + 6)$	from state s_2^2 with input I_2 at time $\mathbf{t}$, to state s_3^2 with output O_2 at time $\mathbf{t} + \mathbf{6}$
	3	1	$f_2(I_2, s_3^2, r \geq t + 6) \rightarrow (O_2, s_4^2, r + 1)$	from state s_3^2 with input I_2 at time $\mathbf{r}$, to state s_4^2 with output O_2 at time $\mathbf{r} + \mathbf{1}$
f_3	1	1	$f_3(I_3, s_1^3, x' \geq 0) \rightarrow (\emptyset, s_2^3, x' + 1)$	from state s_1^3 with input I_3 at time $\mathbf{x'}$, to state s_2^3 with no output at time $\mathbf{x' + 1}$
	2	1	$f_3(I_3, s_2^3, t' \geq x' + 1) \rightarrow (\emptyset, s_3^3, t' + 1)$	from state s_2^3 with input I_3 at time $\mathbf{t'}$, to state s_3^3 with no output at time $\mathbf{t' + 1}$
	3	1	$f_3(I_3, s_3^3, r' \geq t' + 1) \rightarrow (\emptyset, s_4^3, r' + 1)$	from state s_3^3 with input I_3 at time $\mathbf{r'}$, to state s_4^3 with no output at time $\mathbf{r' + 1}$

Table B.5: Formal Component Representation in Our Proposed Formalism

inition 14 we obtain constraints for the values of x, t, r and x', t', r' respectively. The constraints on x, t, r derive from the fact that the first call to function f_2 has to take place after at least one call to f_1 has completed and produced output, since f_2 requires output from f_1. Similarly for f_3, the first call has to take place after f_2 has produced *at least one* output. Furthermore, the average time spent by messages in the connector queues is considered. The average time in queue is obtained from the meta-simulation validation layer. Assuming that the average times spent in the connector queues are $\Delta w_1 = 2, \Delta w_2 = 3, \Delta w_3 = 1$ and $\Delta w_1' = 4, \Delta w_2' = 3, \Delta w_3' = 2$ for f_2 and f_3 respectively, the most trivial constraints that can be derived are:

$$x \geq \Delta w_1, t \geq x + 11, t \geq 8 + \Delta w_2, r \geq t + 6, r \geq 12 + \Delta w_3 \tag{B.1}$$

$$x' \geq x + 11 + \Delta w_1', t' \geq x' + 1, t' \geq t + 6 + \Delta w_2', r' \geq t' + 1, r' \geq r + 1 + \Delta w_3' \tag{B.2}$$

Next, the constraints are solved by a constraint solver. Assume that a solution is:

$$(x = 8, t = 19, r = 25), (x' = 23, t' = 28, r' = 29).$$

For the reference functions f_i^*, the constraint solver returns the same solution for (x^*, t^*, r^*) and (x'^*, t'^*, r'^*):

$$(x^* = 8, t^* = 19, r^* = 25), (x'^* = 23, t'^* = 28, r'^* = 29).$$

Simulation

The above solutions dictate the interleaved execution schedules of the function calls. Interleaved execution schedules are obtained for both composition and reference composition. For this simple model in which the component definitions are almost the same with the exception of some attributes, the interleaved schedules presented in Figure B.3(a) and Figure B.3(b) are the same. Each interleaved execution is represented as a

$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(a) Composition

$f_1^*(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1^*(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2^*(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1^*(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2^*(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3^*(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2^*(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3^*(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3^*(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(b) Reference Composition

Figure B.3: Interleaved Execution Schedules

Labeled Transition System, $L(M)$ and $L(M^*)$ respectively, as shown in Figure B.4.

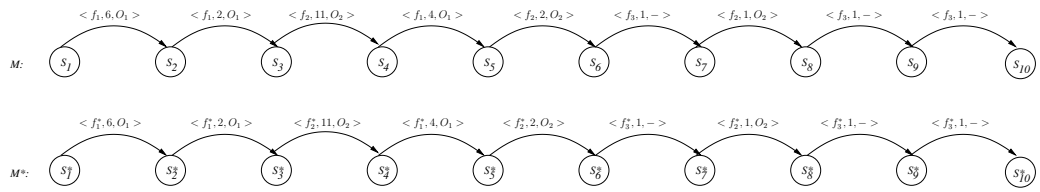


Figure B.4: LTS Representation of Model Execution

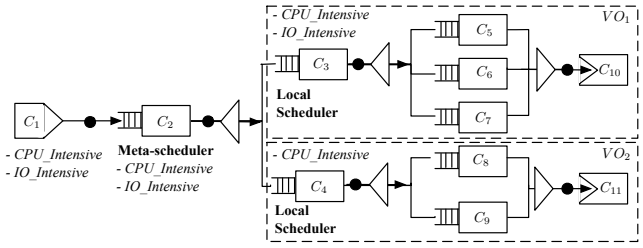
Validation

In the *Validation* step, *strong* equivalence between $L(M)$ and $L(M^*)$ is validated using the BISIMULATOR equivalence checker, part of the CADP toolset. For this simple example, the BISIMULATOR returns `true`. As such, there is no need to validate a possible *weak* equivalence by calculating the semantic metric relation V_ϵ.

Appendix C

Component-based Modeling of a Grid System

The conceptual model of a grid system with two virtual organizations (VO) sharing a grid meta-scheduler job queue [42] is shown in Figure C.1(a). Each virtual organization consists of a local job scheduler and different types of computational resources. Assume the meta-scheduler accepts CPU and I/O intensive jobs that can be serviced by VO_1, whereas VO_2 only services CPU intensive jobs.



(a) Conceptual Model

```
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))
```

(b) Production String using Base Components

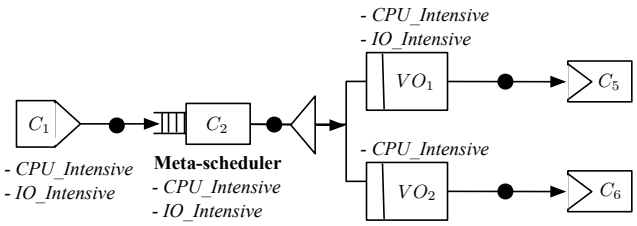
Figure C.1: A Grid Computing System Composed using Base Components

C.1 Conceptual Model Definition

The simulation developer inputs the conceptual model by drag-and-drop icons of Queueing Networks base components on the GUI drawing panel and subsequently connecting them using One-to-One, Fork, and Join connectors as shown in Figure C.1(a).

Conceptual Model Definition using Reused Model Components

To facilitate the reuse of previously developed models, reused model components can be employed when drawing the conceptual model. For example, VO_1 and VO_2 could have been developed earlier and saved into the model repository as model components to be reused as shown in Figure C.2(a).



(a) Conceptual Model

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

(b) Production String using Model Components

Figure C.2: A Grid Computing System Composed using Reused Model Components

C.2 Syntactic Composability Verification

The production string for the grid system conceptual model is presented in Figure C.1(b). The production string is accepted by the queueing network application domain composition grammar and thus syntactic composability is verified.

Syntactic Composability Verification with Reused Model Components

The corresponding production string for the model in Figure C.2(a) is shown in Figure C.2(b). The composition grammar caters for reused model components, as shown in Figure 4.6. As such, the above production string is accepted by the composition grammar parser and the model is considered syntactically correct.

C.3 Model Discovery and Selection

Following syntactic composability verification, each individual component is discovered based on individual component queries provided by the simulation developer. For simplicity, we showcase the discovery of base components. In order to meaningfully rank candidate components from the repository, we calculate the Matching Index for all pairs of component query and repository component description, as discussed in Chapter 5.

We perform several optimizations to reduce the number of repository components considered in the discovery process. When discovery of base components is performed, we consider only repository components of the same *type* as the base components in the conceptual model. The *type* of a component is an example of a mandatory attribute, which needs to be properly defined in the COSMO ontology. For example, when discovery is performed for component C_2 in Fig. 8.9, only components of type $Server$ (or additional sub-types according to the COSMO ontology) are considered. The CoDES component repository contains `1958` base and model components for the Queueing Networks application domain, from which `1949` are model components that contain between 4 and 10 base components connected in different ways, and 9 are base components, 3 each for the types *Source*, *Server*, and *Sink* respectively. This repository models a real-life scenario where we expect the number of base components per applica-

tion domain to be significantly smaller than the number of developed component-based models. Thus, when a query is performed for component C_2, the number of candidate components is reduced to 3, with a decrease of `99.99%`. When model components are discovered, we reduce the state space by considering only the repository model components whose input and output constraints match those that can be deduced from the conceptual model.

Table C.1 presents a sample query for component C_2. The meta-scheduler component C_2 must service two types of jobs, as shown in the Input/Output constraints for the query component C_2. Matching on the required attributes returns a matching index of $MI_r = 1$, since the type attribute matches exactly and all relevant words in the description of the query component are found in the repository component. Only the $origin$ and $destination$ are close matches for the input and output constraints, resulting in $MI_b = 0.25$.

	Query Component (C_2)	**Repository Component** (C)	**Matching Index**
Mandatory Attributes	**type:** Server **description:** server, two classes of jobs	**type:** Server **description:** single-server	$MI_r = 1$
Specific Attributes	-	-	$MI_a = 0$
Behavior: Input/Output Constraints	Input: I_1, I_2, Output: O_1, O_2 $IConstraints = \{I_{1C_1}, I_{2C_1}\}$, $OConstraints = \{O_{1C_1}, O_{2C_1}\}$ I_{1C_1} = { *class = CPU_Intensive, origin = TwoClassSource* } I_{2C_1} = { *class = IO_Intensive, origin = TwoClassSource* } O_{1C_1} = { *class=CPU_Intensive, destination = SingleUnitServer* } O_{2C_1} = { *class=IO_Intensive, destination = SingleUnitServer* }	Input: I_1, Output: O_1 $IConstraints = \{I_{1C_1}\}$, $OConstraints = \{O_{1C_1}\}$ I_{1C_1} = { *type = double, range = 10:35,* *origin = Source|SingleUnitServer* O_{1C_1} = { *type = double, range =10:35,* *destination = SingleUnitServer|Sink*}	$MI_b = \frac{0.5+0.5+0.5+0.5}{2+2+2+2} = 0.25$
			$\mathbf{MI(C_2, C) = 0.625}$

Table C.1: Base Component Query Information

The discovery process continues in a similar manner for all base components in the conceptual model. Assume that key meta-component information for repository components C_1, C_2, and C_4 is shown in Table 6.1. As it can be seen, component C_4 services a single class of jobs, namely $CPU\_Intensive$, whereas C_2 services $IO\_Intensive$ jobs as well. Similarly, components C_3, C_5, C_6, C_7 and C_{10} will serve two classes of jobs, whereas C_4, C_8, C_9, and C_{11} will serve only $CPU\_Intensive$ jobs.

Assume that all components are successfully discovered and the meta-component

information is shown in Table C.2. As it can be seen, component C_4 services a single class of jobs, namely $CPU\_Intensive$, whereas C_2 services $IO\_Intensive$ jobs as well. Similarly, components C_3, C_5, C_6, C_7 and C_{10} will serve two classes of jobs, whereas C_4, C_8, C_9, and C_{11} will serve only $CPU\_Intensive$ jobs. Semantic compos-

	$\mathbf{C_1}$	$\mathbf{C_2}$	$\mathbf{C_4}$
Attribute	$noJobsGenerated = 0$ interArrivalTime: exponential(3)	$noJobsServiced = 0$ serviceTime1 : exponential(6) serviceTime2 : exponential(5) $busy = false$	$noJobsServiced = 0$ serviceTime : exponential(2) $busy = false$
Input	-	I_1, constraints: $origin = Source\|Server \ldots$ $class = CPU\_Intensive$	I_1, constraints: $origin = Source\|Server \ldots$ $class = CPU\_Intensive$
	-	I_2, constraints: $origin = Source\|Server \ldots$ $class = IO\_Intensive$	-
Output	O_1, constraints: $destination = Server \ldots$ $class = CPU\_Intensive$	O_1, constraints: $destination = Server\|Sink$ $class = CPU\_Intensive$	O_1, constraints: $destination = Server\|Sink$ $class = CPU\_Intensive$
	O_2, constraints: $destination = Server \ldots$ $class = IO\_Intensive$	O_2, constraints: $destination = Server \ldots$ $class = IO\_Intensive$	-
State Machine	$S_1(\Delta interArrivalTime) \xrightarrow{[C_1]} S_1O_1[A_1]$ $S_1(\Delta interArrivalTime) \xrightarrow{[C_2]} S_1O_2[A_2]$	$I_1S_1 \rightarrow S_2[A_1; A_3; A_4]$ $I_2S_1 \rightarrow S_3[A_1; A_3; A_5]$ $S_2(\Delta serviceTime1) \xrightarrow{C_1} S_1O_1[A_2]$ $S_3(\Delta serviceTime2) \xrightarrow{C_2} S_1O_2[A_2]$ $\ldots$	$I_1S_1 \rightarrow S_2[A_1; A_3]$ $S_2(\Delta serviceTime) \rightarrow S_1O_1[A_2]$
	$[A_1] = noJobsGenerated++;$ $[C_1] = noJobsGenerated\%2 == 0;$ $[C_2] = noJobsGenerated\%2 == 1;$	$[A_1] = (busy = true);$ $[A_2] = (busy = false);$ $[A_3] = noJobsServiced++;$ $[A_4] = class = C_1;$ $[A_5] = class = C_2;$ $[C_1] = class == C_1;$ $[C_2] = class == C_2;$	$[A_1] = (busy = true);$ $[A_2] = (busy = false);$ $[A_3] = noJobsServiced++;$

Table C.2: Meta-component Representation of Discovered Components

ability validation is performed when the discovery process completes successfully.

C.4 Semantic Composability Validation

C.4.1 Validation of General Model Properties

The first step in the validation of semantic composability ensures that the component communication is meaningful. The compatibility with respect to input and output data between connected components is calculated in a similar manner to MI_b using data from the COSMO ontology, and results in a Composability Index $CI = 1$ [125].

In Concurrent Process Validation, the composition is formally specified using concurrent processes with instantaneous transitions and generic output abstracted to a single message type. This is possible because the component communication has been previously validated. This results in a significant state space reduction by a factor that would otherwise have been a power of the number of states per component and number of components (see Table 8.6). Figure C.3 and Figure C.4 show the Promela model for this example. The SPIN model checker validates this example and the concurrent process validation returns a positive answer.

The Meta-Simulation layer next validates the composition execution through time. Safety and liveness properties are validated over time from a practical simulation perspective. Safety is specified by the simulation developer through validity points that describe the data permitted through certain points in the composition topology. Here, a validity point for the connection between C_8 and C_{11} could be $VP_1 = d_1\{origin = Server, destination = Sink, class = CPU\_Intensive\}$, but more expressive validity points could be permitted. In the validation of liveness, a transient predicate for component C_4 could be $transient(C_4) = \{noJobsServiced == 5\}$. In our example, the transient predicate for component C_4 imposes a QoS on the local scheduler to serve at least five $CPU\_Intensive$ jobs in order to be considered alive. A number of 10 meta-simulation runs are performed and the composed model is considered `invalid` because component C_4 does not satisfy its transient predicate. This is because VO_2 does not get to serve enough $CPU\_Intensive$ jobs.

C.4.2 Formal Validation of Model Execution

Since the composed model did not pass the previous stage, formal validation of model execution is not performed.

```
1  mtype {Job};
2  chan to11 = [10] of {mtype};
3  chan to10 = [10] of {mtype};
4   ...
5  chan from2 = [10] of {mtype};
6  chan from1 = [10] of {mtype};
7  typedef forkChannels{
8          chan outgoing[10] = [10] of {mtype}
9  }

11 typedef joinChannels{
12         chan incoming[10] = [10] of {mtype}
13 }

15 hidden byte sourceIAMax = 10;
16 byte sourceIATime;
17 byte noJobsSource;

19 proctype CON_ONE_TO_ONE(chan in, out){
20 byte noJobs;
21 do
22   :: in ? Job -> out ! Job; noJobs++;
23 od
24 }

26 proctype CON_FORK1(int noChannels; chan in, out0,out1,out2){
27  int currentChan = 0;
28  forkChannels chans;
29 chans.outgoing[0] = out0;
30 chans.outgoing[1] = out1;
31 chans.outgoing[2] = out2;
32 do
33          :: in ? Job -> chans.outgoing[currentChan % noChannels] ! Job; currentChan =
34          currentChan + 1; od
35 }
```

Figure C.3: Specification of a Grid System in Promela

```
1  proctype CON_FORK(int noChannels; chan in,
2  out0,out1){  int  currentChan = 0;
3   forkChannels  chans;
4  chans.outgoing[0]  = out0;
5  chans.outgoing[1]  = out1;
6  do
7              ::  in  ?  Job  −> chans.outgoing[currentChan % noChannels] ! Job;
8             currentChan = currentChan + 1;  od
9  }
10 proctype CON_JOIN(chan out, in0,in1,in2){    ...}
11 proctype SINK1(int  id;  chan in){
12 S1: {if  ::  in  ?  Job  −> printf("[Sink]  Job  received!\n");goto S1; fi }}

14 proctype SERVER3(int id; chan   in ,  out){
15 byte  noJobsQueue;
16 S1: {
17 if
18           ::  in  ?  Job  −> printf("[Server]  Job  received!\n");
19                    noJobsQueue++;goto S2;
20  fi  }
21 S2: {
22 if
23           ::  out  !  Job  −> progress: printf("[Server]  Job  sent!  \n");
24           if
25            ::  noJobsQueue==1−>              noJobsQueue−−;goto S1;
26            ::  noJobsQueue!=1−>              noJobsQueue−−;goto S2;
27           fi  fi    } }

29 proctype SOURCE1(int id; chan out){
30 do
31           ::  (sourceIATime == sourceIAMax) −>
32           sourceIATime = 0;
33           if
34           ::  out  !  Job  −> progress :  printf("[Source]  Job  sent\n"); fi  od }

36 proctype SourceCounter(){
37 do
38           ::  (sourceIATime < sourceIAMax) −> sourceIATime++;
39 od
40 }

42  init {
43   run  SourceCounter();
44 run  SINK1(11,to11);
45 run  SINK1(10,to10);
46 run  SERVER3(9,to9, from9);
47 run  SERVER3(8,to8, from8);
48 run  SERVER3(7,to7, from7);
49 run  SERVER3(6,to6, from6);    ...
50  }
```

Figure C.4: Specification of a Grid System in Promela (continued)

Appendix D

Component-based Modeling of a Tank vs SoldierTroop System

D.1 Conceptual Model Definition

The simulation developer inputs the conceptual model by drag-and-drop icons of a Tank and a SoldierTroop base components on the GUI drawing panel and subsequently connecting them using One-to-One connectors as shown in Figure D.1. As it can be seen, this results in a closed system with a feeback loop going in the SoldierTroop base component.

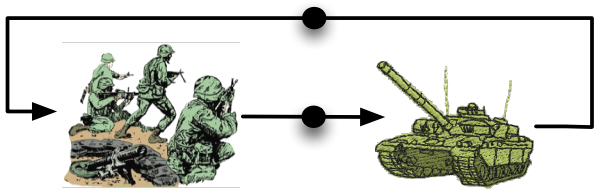


Figure D.1: Tank vs Soldier Troop Training Scenario

D.2 Syntactic Composability Verification

The conceptual model presented above is syntactically verified by the *Syntax Verifier* module. First, the module checks that no component port is left unconnected. Next, a production string is generated to describe the conceptual model. The production string is a linear arrangement of the components' types according to their position on the graphical screen. The production string for this model is:

$$TankMTS = Tank\ ConO\ SoldierTroop\ ConO\ Tank$$

The model is syntactically verified if the production string is accepted by the Military Training composition grammar from Figure 8.4. In the CoDES implementation, the composition grammar is first parsed using our implemented Earley parser. Next, the production string is verified by the parser. The parser accepts the production string and thus the model is syntactically correct.

D.3 Model Discovery and Selection

Following syntactic composability verification, each individual component is discovered based on individual component queries provided by the simulation developer. Assume that the components have been successfully discovered and their meta-components are presented in Table D.1.

Entity	Attribute	Input	Output	State Machine
$\mathbf{tank_1}$	$health = 100$ $range = 7$ $ammo = 50$ movingTime: exponential(5) shootingTime: exponential(4) $usableThreshold = 20$ $positionX = 20$ $positionY = 15$ $speed = 10$ $team = red$ ... $transient(tank_1) : (ammo == 49)$	I_1, constraints: $class = PositionInfo$ $origin = SoldierTroop$ I_2, constraints: $class = InputFire$ $origin = SoldierTroop$	O_1, constraints: $class = PositionBroadcast$ $destination = SoldierTroop$ O_2, constraints: $class = OutputFire$ $destination = SoldierTroop$	$I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_2A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_2A_4$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_1S_1 \xrightarrow{C_3} O_1S_1$ $null \quad S_2(\Delta movingTime) \rightarrow O_1S_1A_1$ C_1 : *no opponents in range* C_2 : *at least one opponent in range* $C_3 = health < usableThreshold$ A_1 : *modify position* A_2 : *modify target position* A_3 : *modify position, health* A_4 : *modify target position, health*
$\mathbf{troop_1}$	$health = 100$ $range = 2$ $ammo = 20$ movingTime : exponential(3) shootingTime : exponential(2) $usableThreshold = 40$ $positionX = 40$ $positionY = 45$ $speed = 3$ $team = blue$ ... $transient(troop_1) : (ammo == 49)$	I_1, constraints: $class = PositionInfo$ $origin = Tank$ I_2, constraints: $class = InputFire$ $origin = Tank$	O_1, constraints: $class = PositionBroadcast$ $destination = Tank$ O_2, constraints: $class = OutputFire$ $destination = Tank$	$S_0 \rightarrow O_1S_1$ $I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ C_1 : *no opponents in range* C_2 : *at least one opponent in range* $C_3 = health < usableThreshold$ A_1 : *modify position* A_2 : *modify target position* A_3 : *modify position, health* A_4 : *modify target position, health*

Table D.1: Meta-component Information for Tank vs. SoldierTroop Scenario

D.4 Semantic Composability Validation

D.4.1 Validation of General Model Properties

Concurrent Process Validation

The *Concurrent Process Validation* layer validates the component *coordination* of the composed model. This layer guarantees that safety, liveness, as well as deadlock freedom hold for all possible interleaved executions of *instantaneous* transitions of the composed simulator abstracted as a composition of concurrent processes. A composed model is invalid if it is found to be deadlocked, or if any of the components invalidate their safety or liveness properties. The behavior of each meta-component modeled as a state machine is translated into a logical specification using a logic converter module. Different converters are developed for each application domain and targeting various logical properties. The converter takes as inputs the meta-components and the compo-

sition topology. The result is a specification describing the composition together with an expression of the safety and liveness properties. To prevent state explosion, each component state machine is reduced by considering only communication states and attributes that influence state transitions. The actions of non-communicating states are abstracted as a single atomic operation. Similarly, time is not modeled and transitions are considered instantaneous.

Figure D.2(a) shows a possible translation of the component state machine into a Promela specification. Each state is transformed into a Promela label, and the label includes input and/or output actions as specified by the meta-component behavior, as well as conditions on attribute values and attribute modifications. Transitions between states are instantaneous. Thus, time attributes such as $\Delta shootingTime$ and $\Delta moovingTime$ from Table D.1 are ignored. Each type of connector is defined as a Promela process. For example, process *CON_ONE_TO_ONE* on line 3 describes the one-to-one connector. The fork and join connectors are not part of this composition and as such are omitted. In the `init` method on line 20, communication channels are assigned to the connectors and components according to their connection topology. Similar to the behavior of connectors in the real system, communication in our Promela specification is asynchronous. Liveness is specified using `progress` labels such as the one on line 7, and safety is specified using `assert` statements. Next, the Promela specification is validated by the Spin model checker [14].

Discussion This example has led to some interesting observations on the translation from a component state machine to a feasible Promela specification. Previously for the Queueing Networks Application Domain, the non data-driven state machines could be almost exactly transformed into Promela and the process was easily automated [119]. However, when data-driven component state machines are used, the process is not easily automated. For example, if we were to interpret component coordination strictly

```
1 mtype {MSG}; chan to1 = [10] of {mtype}; ...
2 proctype CON_ONE_TO_ONE(chan in, out)
3 {do :: in ? MSG -> out ! MSG; od}
4
5 proctype TANK(byte id; chan in, out){
6 S1: atomic{ if :: in ? MSG ->
7  if :: out ! MSG -> progress: printf("MSG sent\n");
8                    goto S1; fi fi}}
9
10 proctype SOLDIERTR(byte id; chan in, out){
11 bit initial = 1;
12 S0: atomic{ if
13 :: (initial == 1) -> initial = 0;
14  if :: out ! MSG -> goto S1; fi fi}
15 S1: atomic{ if
16 :: in ? MSG ->
17 if :: out ! MSG -> progress: printf("MSG sent\n");
18                    goto S1; fi fi
19 }}
20 init{ run TANK(1, to1, from1);
21 run SOLDIERTR(2, to2, from2);
22 run CON_ONE_TO_ONE(from1, to2);
23 run CON_ONE_TO_ONE(from2,to1); }
```

(a) Simple Promela Specification

```
1 proctype SOLDIERTR(byte id, health, ..., posX, posY; chan in, out)
2
3 {bit initial = 1; byte posXFire, posYFire;
4 byte msgPosX, msgPosY, auxX, auxY, auxDistance,...;
5 S0: atomic{
6 if  :: (initial == 1) -> initial = 0;
7  if :: out ! MSG_POS -> goto S1; fi fi}
8 S1: atomic{ if atomic{if
9 :: in ? MSG_FIRE, msgPosX, msgPosY -> health = health - 10;
10
11  if :: health < health_threshold ->
12       if :: out ! MSG_DIE -> goto end; fi
13  :: else
14     if :: out!MSG_POS, posX, posY -> progress: printf("MSG sent\n");
15
16      goto S1;fi
17   fi
18 :: in ? MSG_DIE -> out ! MSG_DIE;goto end;
19 #GPS coord
20 :: in ? MSG_POS, msgPosX, msgPosY ->
21  if :: !(msgPosX<posX-range||msgPosX>posX+range ||msgPosX<posY-range
22       ||msgPosY<posY+range)->
23       if :: ammo>0->out!MSG_FIRE,msgPosX,msgPosY; ammo--; goto S1; fi
24  :: else -> auxDistance = distance;
25 :: msgPosX<posX->auxX = msgPosX+range;
26 :: else -> auxX = msgPosX - range; fi} ...
27 //similar to calc nxt position
28 if #broadcast position
29 :: out ! MSG_POS, posX, posY -> goto S1; fi
30 fi fi }
31 end: skip; }
32 init{
33 run TANK(1, 100, 20, 5, 40, 45,  to1, from1);
34 run SOLDIERTR(2, 100, 10, 5, 15, 20, to2, from2);
35 run CON_ONE_TO_ONE(1, from1, to2);
36 run CON_ONE_TO_ONE(2, from2,to1); }
```

(b) Detailed Promela Specification

Figure D.2: Tank vs SoldierTroop in Promela

from a message passing perspective, the resulting Promela specification would be that presented in Figure D.2. This type of interpretation is easily automated and focuses only on component coordination. However, it lacks expressivity and any coordination logic. On the other hand, if we were to exactly transform the component state machines from their COML specification into Promela like in Figure D.2(b) for the $troop_1$ component, we would obtain a more exact description of the attack but it is difficult to automate the translation process. In this example, we represent the composition according to Figure D.2(b) and consider finding a middle ground between the two approaches part of our future work. The Spin model checker validates the specification and the validation process can proceed to the next layer.

Meta-Simulation Validation

The meta-simulation layer validates if the logical properties demonstrated previously

hold through time. Our implementation translates the complete state machine of each component into a Java class hierarchy. Attributes and their values provided by the user, state transitions, and time are modeled. Next, we construct a meta-simulation of the composed model using the translated classes. During the meta-simulation run, sampling is performed for attributes that require so. This is the case especially for time attributes such as shooting time and moving time. For example, as shown in Table D.1, the shooting time $\Delta shootingTime$ for component $tank_1$ is sampled from an exponential distribution with a $mean$ of 4. The distribution type and mean values, as well as the initial position on the grid ($positionX$ and $positionY$) and the initial ammunition ($ammo$), are examples of attribute values provided by the user. Since sampling is performed, the meta-simulation is run for $N = n * noSampling$ times, where n is the total number of components and $noSampling$ is the total number of locations where sampling is done. If any of the properties does not hold in the meta-simulation runs, the composition is declared invalid.

Two important logical properties to be validated through time are safety and liveness. From a practical perspective, we consider safety to mean that components do not produce invalid output. The simulator developer specifies the desired valid output by providing *validity points* at various connection points in the composition. A validity point contains semantic description of data that must pass through its assigned connection point. For example, one validity point for the data that passes through the feedback one-to-one connector in Figure D.1 could be $VP_1 = d_1\{origin = SoldierTroop, destination = Server, position.X\{range = 20; 40, type = int\}\}$, showing that the new position for component $troop_1$ is calculated properly. A safety error is issued if anytime during the meta-simulation run semantically incompatible data according to the component-oriented ontology passes through the connection point.

Liveness is validated by considering a *transient* predicate assigned to each compo-

nent. The value of the transient predicate is ideally provided by the component creator in the meta-component as shown in Table D.1. Its initial value is `false`. A component is considered *alive* if its liveness observer has evaluated the transient predicate to *true* and then to *false* in an interval of time smaller than the specified timeout. For example, the transient predicate for component $tank_1$ could be $transient(tank_1) = (ammo == 49)$. This guarantees that the tank must shoot at least twice for it to be considered alive. A *liveness observer* is attached to each component and is notified every time the attributes involved in a transition change values. Once the meta-simulation layer returns a positive value, the validation process can proceed to the next layer.

D.4.2 Formal Validation of Model Execution

In step 2, a model M composed of $tank_1$ and $troop_1$ is validated by comparison with a reference model M^* consisting of reference components $tank^*$ and $troop^*$. A reference component is a generic, desirable representation of a base component ideally provided by domain experts when the new application domain is added to the framework. Ideally, the reference components should describe what the system experts consider to be the desirable base component behavior. It should be generic in the sense that their description lacks any real data values. Throughout the validation process, the generic reference components attributes will be instantiated using the same attribute values used by the corresponding components in the composed model M. The attribute correspondence is established by using the COSMO ontology. For the military training application domain, we assume the reference component $troop^*$ to be the same as component $troop_1$ from Table 8.1. Let component $tank^*$ state machine be the one presented in Table D.2. Notice that the difference between $tank^*$ and $tank_1$ is in the missing state S_2. This is because $tank^*$ implements a direct attack tactic whereas $tank_1$ implements a shoot and scoot tactic.

Entity	Data	State Machine
$\mathbf{tank}^*$	**Input** I_1, constraints: $class = PositionInfo$ $origin = SoldierTroop$	$I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_4$
	I_2, constraints: $class = InputFire$ $origin = SoldierTroop$	$I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_1S_1 \xrightarrow{C_3} O_1S_1$
	Output O_1, constraints: $class = PositionBroadcast$ $destination = SoldierTroop$	C_1 : *no opponents in range* C_2 : *at least one opponent in range* $C_3 = health < usableThreshold$ A_1 : *modify position* A_2 : *modify target position*
	O_2, constraints: $class = OutputFire$ $destination = SoldierTroop$	A_3 : *modify position, health* A_4 : *modify target position, health*

Table D.2: Reference Component State Machine

Our formal validation layer is divided into five steps, namely (i) *Formal Component Representation* in which component state machines are translated into our proposed time-based formalism, (ii) *Unfolding and Sampling* in which time attribute values are sampled, (iii) *Mathematical Composability* in which the mathematical composability of functions is validated, (iv) *Representation of Model Execution* in which the execution of the composed model is represented as a Labelled Transition System [114], and (v) *Bisimulation Validation* in which the execution of model M is validated against the execution of model M^* [122].

In Definition 12, components $tank_1$ and $troop_1$ are represented formally as mathematical functions f_1 and f_2 respectively. Model M is described formally as $M = \{(f_1, f_2),(f_2, f_1)\}$. Conversely, $tank^*$ and $troop^*$ are represented formally as f_1^* and f_2^* respectively and their composition is represented formally as $M^* = \{(f_1^*, f_2^*), (f_2^*, f_1^*)\}$. In the first four steps, M and M^* are transformed in a format that facilitates meaningful comparison. In the following we present the translation process for f_1 and f_2. The process for f_1^* and f_2^* is exactly the same.

Formal Component Representation

The state machine for component $tank_1$ is translated to a formal component representation specified by f_1 as

$$f_1 : \{I_1, I_2\} \times S_1 \times T_1 \rightarrow \{O_1, O_2\} \times S_1 \times T_1,$$
$$f_1(I_1, s_i, t) \rightarrow (O_1, s_i', t + \Delta t),$$
$$f_1(I_1, s_i, t) \rightarrow \quad (O_2, s_i', t + \Delta t),$$
$$f_1(I_2, s_i, t) \rightarrow \quad (O_1, s_i', t + \Delta t),$$
$$f_1(I_2, s_i, t) \rightarrow (O_2, s_i', t + \Delta t),$$
$$f_1(null, s_i, t) \rightarrow (O_1, s_i', t)$$

where Δt is sampled from a specific distribution (either the distribution for $movingTime$ or $shootingTime$) and the function is re-called until $t > T$, where the simulation runs for time $T = 400$ wall clock units.

Unfolding and Sampling

As it can be seen, the above expression for f_1 is not useful because during a simulation run, t and Δt have specific values. In this step, we unfold the function definition for $\tau = 5$ times and sample the values for Δt from $\Delta movingTime$ or $\Delta shootingTime$, using mean values provided by the user. For component $tank_1$ described formally as f_1, Δt takes values as necessary from the sampled values of $movingTime, exponential(mean = 5) = \{20, 40, 70\}$, and $shootingTime, exponential(mean = 4) = \{10\}$. The values of f_1 and f_2 are presented in Table D.3. f_2 is described first, because according to the state machine in Table D.3, it is the $troop_1$ component that will initiate the communication.

Mathematical Composability

Next, the function composability is validated in the *Mathematical Composition* step. Following Definition 14, we obtain constraints for the values of var_1, var_2,var_3, var_4 and $var_{21}, var_{22}, var_{23}$ respectively.

	Unfold	Δt	Formula		
f_2	1	-	$f_2(\emptyset, s_1^2, 0 \geq 0)$	$\rightarrow$	$(O_1, s_2^2, 0)$
	2	20	$f_2(I_1, s_2^2, var_1 \geq 0)$	$\rightarrow$	$(O_1, s_3^2, var_1 + 20)$
	3	40	$f_2(I_2, s_3^2, var_2 \geq var_1 + 20)$	$\rightarrow$	$(O_1, s_4^2, var_2 + 40)$
	4	10	$f_2(I_1, s_4^2, var_3 \geq var_2 + 40)$	$\rightarrow$	$(O_2, s_5^2, var_3 + 10)$
	5	80	$f_2(I_2, s_3^2, var_4 \geq var_3 + 10)$	$\rightarrow$	$(O_1, s_6^2, var_4 + 80)$
f_1	1	50	$f_1(I_1, s_1^1, var_{21})$	$\rightarrow$	$(O_1, s_2^1, var_{21} + 50)$
	2	10	$f_1(I_1, s_2^1, var_{22} \geq var_{21} + 50))$	$\rightarrow$	$(O_2, s_3^1, var_{22} + 10)$
	3	3	$f_1(\emptyset, s_3^1, var_{22} + 10)$	$\rightarrow$	$(O_1, s_4^1, var_{22} + 13)$
	4	30	$f_1(I_1, s_4^1, var_{23} \geq var_{22} + 13)$	$\rightarrow$	$(O_2, s_5^1, var_{23} + 30)$
	5	7	$f_1(\emptyset, s_5^1, var_{23} + 30)$	$\rightarrow$	$(O_1, s_6^1, var_{23} + 37)$

Table D.3: Formal Component Representation

The constraints on var_{21}, var_{22}, var_{23} derive from the fact that the first call to function f_1 has to take place after *at least one* call to f_2 has completed and produced output, since f_1 requires output from f_2. Because there exists a *feedback loop* between f_2 and f_1, the second call for f_2 at time var_1 has to take place at least after the first call to f_1, resulting in the $var_1 \geq var_{21} + 50 + w_{21}$, where w_{21} is the average time spent in the connector queue from f_2 to f_1. From a realistic perspective, we also consider the average time spent by messages in the connector queues, which is obtained from the meta-simulation validation layer. Assuming that the average times spent in the connector queues are $\Delta w_{12} = 2$, $\Delta w_{21} = 3$ for the connector between f_1 and f_2 and vice-versa, the most trivial constraints that can be derived are:

$var_1 \geq 0, var_1 \geq var_21 + 50 + \Delta w_{12},$

$var_2 \geq var_1 + 20, var_2 \geq var_{22} + 10 + \Delta w_{12},$

$var_3 \geq var_2 + 40, var_3 \geq var_{22} + 13 + \Delta w_{12},$

$var_4 \geq var_3 + 10, var_4 \geq var_{23} + 30 + \Delta w_{12}.$

$var_{21} \geq 0 + \Delta w_{21},$

$var_{22} \geq var_{21} + 50, var_{22} \geq var_1 + 20 + \Delta w_{21},$

$var_{23} \geq var_{22} + 13, var_{23} \geq var_2 + 40 + \Delta w_{21}.$

Next, the constraints are solved by the Choco constraint solver [23]. Assume that a solution is:

$$f_2 : (var_1 = 56, var_2 = 91, var_3 = 131, var_4 = 166),$$
$$f_1 : (var_{21} = 4, var_{22} = 79, var_{23} = 134).$$

The same process is applied for reference functions f_i^* using the same sampled values and average waiting times. However, the set of constraints and the number of variables are different because of the different implementation for component $tank_1$.

$$var_1^* \geq 0, var_1^* \geq var_{21}^* + 50 + \Delta w_{12},$$
$$var_2^* \geq var_1^* + 20, var_2^* \geq var_{22}^* + 10 + \Delta w_{12},$$
$$var_3^* \geq var_2^* + 40, var_3^* \geq var_{23}^* + 30 + \Delta w_{12},$$
$$var_4^* \geq var_3^* + 10, var_4^* \geq var_{24}^* + 20 + \Delta w_{12}.$$
$$var_{21}^* \geq 0 + \Delta w_{21},$$
$$var_{22}^* \geq var_{21}^* + 50, var_{22}^* \geq var_1^* + 20 + \Delta w_{21},$$
$$var_{23}^* \geq var_{22}^* + 10, var_{23}^* \geq var_2^* + 40 + \Delta w_{21},$$
$$var_{24}^* \geq var_{23}^* + 30, var_{24}^* \geq var_3^* + 10 + \Delta w_{21},$$
$$var_{25}^* \geq var_{24}^* + 20, var_{25}^* \geq var_4^* + 80 + \Delta w_{21}.$$

The constraint solver returns the following solution:

$$f_2^* : (var_1 = 56, var_2 = 91, var_3 = 166, var_4 = 201),$$
$$f_1^* : (var_{21} = 4, var_{22} = 79, var_{23} = 134, var_{24} = 179, var_{25} = 284).$$

Representation of Model Execution

Interleaved execution schedules are next obtained for both composition and reference composition, in Figure D.3(a) and Figure D.3(b). Each interleaved execution is rep-

$f_1(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(a) Composition

$f_1^*(\emptyset, s_1^1, 0) \rightarrow (O_1, s_2^1, 6)$
$f_1^*(\emptyset, s_2^1, 6) \rightarrow (O_1, s_3^1, 8)$
$f_2^*(I_2, s_1^2, 8) \rightarrow (O_2, s_2^2, 19)$
$f_1^*(\emptyset, s_3^1, 8) \rightarrow (O_1, s_4^1, 12)$
$f_2^*(I_2, s_2^2, 19) \rightarrow (O_2, s_3^2, 25)$
$f_3^*(I_3, s_1^3, 23) \rightarrow (\emptyset, s_2^3, 24)$
$f_2^*(I_2, s_3^2, 25) \rightarrow (O_2, s_4^2, 26)$
$f_3^*(I_3, s_2^3, 28) \rightarrow (\emptyset, s_3^3, 29)$
$f_3^*(I_3, s_3^3, 29) \rightarrow (\emptyset, s_4^3, 30)$

(b) Reference Composition

Figure D.3: Interleaved Execution Schedules

resented as a Labeled Transition System, $L(M)$ and $L(M^*)$ respectively, as shown in Figure 7.3. Each node represents an annotated composition state $S_{j=1,n*\tau}$. Edges are the function calls f_i and f_i^* respectively, and labels are the tuple <*function_name, duration, output*>, where $duration$ represents the function execution time. The labels consider the $duration$ rather than the $time$ moment when the function begins to execute, since the time moments are already ordered through the directed nature of the LTS.

Bisimulation Validation

In the *Validation* step, we check the bisimulation between $L(M)$ and $L(M^*)$ using the BISIMULATOR tool in the CADP toolset [45].

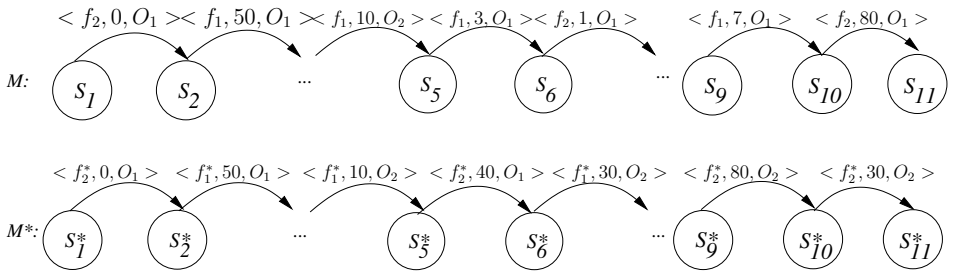


Figure D.4: LTS Representation of Model Execution

It is evident that the two LTS are not strongly equivalent (see the outgoing labels

from S_5, S_6, S_9, S_{10} and S_5^*, S_6^*, S_9^*, S_{10}^* respectively), hence the BISIMULATOR tool returns `false`. Next, we relax the validation constraints by defining a semantic metric relation V with parameter ϵ. V_ϵ considers only semantically related LTS nodes for which our defined semantic distance is smaller than ϵ. A node S_i from $L(M)$ is related to a node S_j^* from $L(M^*)$ iif $d(S_i, S_j^*) \leq \epsilon$. The calculation of d considers (i) the function that is called to exit the two nodes respectively, and (ii) the similarity of the composition states in nodes S_i and S_j^*. The composition state refers to all attribute values for all components in the composition. As such, for attribute names that are the same or similar (according to the COSMO ontology), we consider whether their values are the same or have followed a similar modification trend (e.g. $ammo$ has been decreasing) throughout the unfolding. From the related states set we construct two new LTS, $L_1(M)$ and $L_1(M^*)$ as follows. For each pair of related states (S_i, S_j^*), with $S_i \in L(M)$ and $S_j^* \in L(M^*)$ we add to $L_1(M)$ all pairs (S_i, S_k), where there exists an edge between S_i and S_k in $L(M)$. Similarly, we add to $L_1(M^*)$ all pairs (S_j^*, S_r^*), where there exists an edge between S_j^* and S_r^* in $L(M^*)$. Next, we try to determine the relation between the new $L_1(M)$ and $L_1(M^*)$. We iteratively try possible relations including equivalence, smaller than ($L_1(M)$ included in $L_1(M^*)$), and greater than ($L_1(M^*)$ included in $L_1(M)$).

For this example, we calculate the semantic metric relation V_ϵ for $\epsilon = 0.25$ and obtain the following related nodes: $V_\epsilon = \{(S_i, S_j^*) \| i \neq 5, 6, 9\}$, with $\{\|S_i - S_j^*\|_\sigma = 0.07 | \forall i \neq 5, 6, 9\}$. For these values of V_ϵ we construct two new LTS, $L_1(M)$ and $L_1(M^*)$, by omitting nodes S_5, S_6, and S_9 from $L(M)$. Space constraints prevent us from showing the detailed process here, but it can be seen from the V_ϵ set that $L_1(M)$ is included in $L_1(M^*)$.

Appendix E

Implementation Overview

A high level overview of the CoDES modules is presented in Figure E.1.

Five main packages form the back-end of the CoDES framework, namely `base`, `syntax`, `discovery`, `validation`, and `utils`. The `base` package contains classes that define meta-components and compositions, such as `MetaComponent`, `Composition`, `Connector`, `Attribute`, and `Behavior` among others. The `syntax` package contains an EBNF grammar parser which is an implementation of an Earley parser. The `discovery` package contains helper classes that calculate the degree of similarity between the user query and a repository component. It uses the Jena reasoner to query the COSMO ontology and rank components based on attribute and behavior matching. The `validation` package is separated into four modules that validate: a) data compatibility according to the COSMO ontology; b) correct coordination - the SPIN model checker is employed to validate all interleaved executions of components in the composed model; c) correct computation - a threaded time-based concurrent execution of the composition; d) similarity with respect to a reference model - a Labeled Transition System representation of the composition is used to reason about similarity. The `utils` package contains utility classes for parsing XML files among others. The size of the CoDES back-end prototype is around 20,000 LOC.

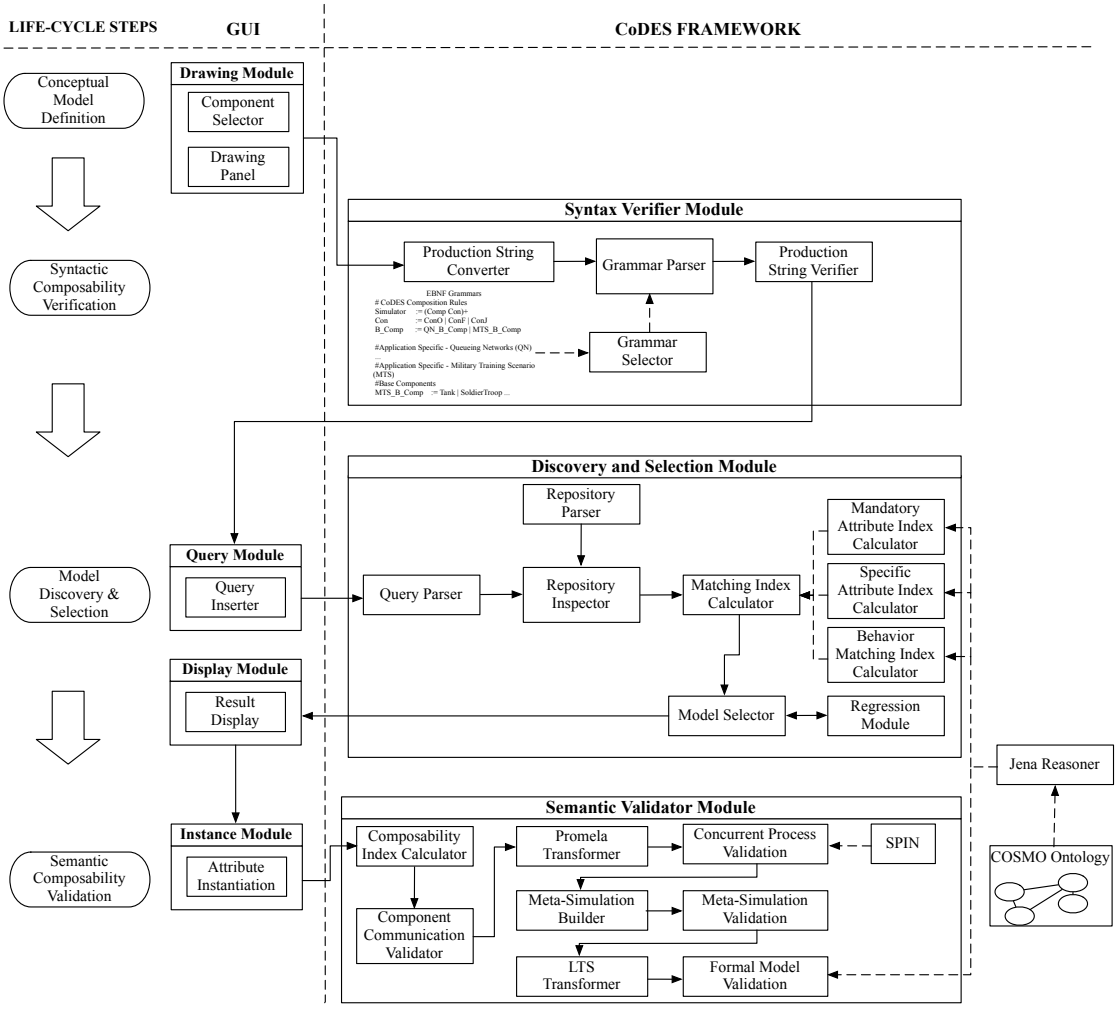


Figure E.1: High Level Overview of the CoDES Implementation