

In Search of the Philosopher's Stone: Simulation Composability Versus Component-Based Software Design

Robert G. Bartholet
David C. Brogan
Paul F. Reynolds, Jr.
Joseph C. Carnahan

Modeling and Simulation Technology Research Initiative
University of Virginia
151 Engineer's Way, PO Box 400740
Charlottesville, VA 22904-4740
434-982-2291|2211|1039|2291
{bartholet|dbrogan|reynolds|carnahan}@cs.virginia.edu

Keywords:
composability, interoperability, reuse, component, software

ABSTRACT: *The simulation community and the software engineering community are actively conducting research on technology that will make it possible to easily build complex systems by combining existing components. Advances in these research areas offer both communities numerous benefits, including reduced development time and the ability to explore a wider space of design alternatives by adding and removing components from existing software systems. In the simulation community this research falls under the umbrella of composability. In the software engineering community it is referred to as component-based software design (CBSD). We show that simulation composability and CBSD are fundamentally the same. Both communities have made significant progress addressing the syntactic, or software connection, issues of composability, but it has been difficult to guarantee that composed components behave meaningfully (described as semantic composability within the simulation community). We demonstrate that although the software engineer's perspective on the composability problem is different, it differs only in terms of semantics and scale. By focusing on the similarities, we will show where the simulation community can gain insight from past and current CBSD research within the software engineering community. Additionally, we will address unique characteristics of simulations, such as the common use of stochastic sampling, time management, and event generation, for providing special opportunities for composability.*

1. Introduction

Within both the simulation and software engineering communities, there has been a flurry of activity in the past decade to ease the burden of implementing complex software systems. The software engineering community is looking at this problem for the general case that includes all software systems, whereby the modeling and simulation (M&S) community is focused on simulations. Interestingly, though both groups attack this problem from a different perspective, both have identified a similar solution framework that provides the ability to build a working, meaningful, complex software system from a set of components. In the software engineering community the solution framework is called component-based software design

(CBSD)¹. In the M&S community, it is (simulation) composability.

Many have theorized about the existence of such a solution framework in the general case, both for software, and its subset, simulations. Contrast this hypothetical solution with the ancient myth of the philosopher's stone, for which many conducted an exuberant, irrational search for an unknown substance that would transmute base metals into gold. While a solution for building meaningful, complex software systems from components may theoretically exist (unlike the philosopher's stone), the limited technological advances to date in achieving this goal may suggest that our initial exuberance in both CBSD and composability was also somewhat irrational. We believe additional advances in building software from

¹ CBSD is also called component-based software engineering (CBSE). In this paper we use CBSD.

components will be achieved, though of a more limited scale and scope than solving this challenge for the general case.

The search for a general composability solution is still a noble one. The ability to build complex software systems from a set of components will offer numerous benefits to both the software and simulation communities. If one can meet the requirements of a complex system by pulling components off the shelf and gluing them together with a reasonable level of effort, then there exists the potential for a significant savings in development time. Additionally if there are several components that meet the same requirement(s), then a wider space of design alternatives can be implemented and tested, providing greater flexibility to the system designers and implementers.

There are characteristics of simulations that can provide purchase in the composability problem. These include time management, stochastic sampling, and event generation. These characteristics can be used to argue that simulation composability is a unique challenge. However, despite the assertion by some that software engineers and simulationists are trying to solve a fundamentally different problem, we will show that the problem is actually the same. The technologies developed to date by each community have achieved strikingly similar results. It follows that the research challenges that lie ahead for both communities are also fundamentally the same. Hence, the simulation community should pay close heed to current and future research within the software engineering community as it provides potential for progress in simulation composability.

The remainder of this paper is structured as follows. In sections 2 and 3, we discuss the state of the art in building complex systems from components in the simulation and software engineering communities respectively. In section 4 we make the case that composability and CBSD are more similar than different. In section 5 we show where software engineering research and technologies provide opportunities that can be applied to simulation composability, and where unique characteristics of simulations may provide unique solutions for the simulation domain. Finally, we draw some conclusions in section 6.

2. Simulation Composability State-of-the-Art

Composability has been defined as the capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements [1]. Interoperability differs from composability because it only requires components to be combined in a meaningful way for a single instance, contrasted with composability which requires the ability to combine and recombine components in other ways to meet new objectives without requiring substantial integration efforts [1]. What exactly constitutes a “substantial integration effort” is subjective. Therefore the delineation between composability and interoperability is admittedly a sliding scale without objective boundaries. In some respects, if strict composability is too hard, interoperability might be good enough, providing a tolerable balance between simulation component reuse and development efforts.

To date, there have been no significant breakthroughs in building practical, composable, simulation frameworks. Most of the progress has been achieved on the theoretical side of composability, most notably complexity results for the component selection problem [2][3][4], and a formal theory of composability [5][6]. Contrast this lack of progress in composability with simulation interoperability, which in the last decade has seen a plethora of technologies emerge to advance the M&S community. Most notably, the development of Distributed Interactive Simulation (DIS) [7], Aggregate Level Simulation Protocol (ALSP) [8], and the High Level Architecture (HLA) [9] have given simulation developers some tools to glue together simulation components into an interoperable federation.

These tools do, however, have two significant, related shortfalls: the inability to guarantee consistency between simulations in the federation, and the inability to provide for component reuse in other federations without significant source code modifications. In other words, while these tools can provide the means for exchanging and maintaining entity state, resolving interactions, managing time, and managing data distribution, they provide only minimal facilities for ensuring the federation is a meaningful simulation. This lack of support often results in simulations that fail to meet requirements because of a lack of consistency caused by fundamental differences in underlying models.

Composability theory explains how an executing federation can provide imperfect results. There are two types of composability, syntactic and semantic [1]. Syntactic composability requires compatible implementation details for all possible compositions. Examples of implementation details include timing mechanisms and interface specifications. Semantic composability requires a meaningful, or valid composition. Informally this definition implies that the assumptions made by each component in a composition remain consistent throughout the execution of the simulation. Both syntactic and semantic composability are necessary for simulation composability.

Historically, syntactic composability has been achievable, especially if components are built to a common engineering framework. Semantic composability, on the other hand, has been a much more difficult undertaking. The potential for some new technologies to allow researchers to gain footholds in solving semantic composability is explored further in section 5.

A recent DoD initiative is the Product Line Approach within the One Semi-Automated Forces (OneSAF) simulation framework. This framework provides an integrated system for planning, generating, and managing a simulation composed of components built to support the framework [10]. OneSAF is an HLA compliant system, so it has all the interoperability tools that come with the HLA specification. Additionally, OneSAF provides a host of tools for composing a simulation from a set of behaviors, entities, units, and environmental models, all architected to conform to the OneSAF framework. However, OneSAF does not contain any ability to enforce assumptions and dependencies bound to a component as it is reused in a different context. The advantage of the OneSAF framework is that the set of models from which to construct a simulation have all been engineered to work within the framework. Therefore, as long as developers stay within the framework, they should have the ability to modify these models to meet new requirements, without concern for the pre-existing code that provides the glue into the framework. The community awaits final evaluation of the OneSAF framework, which will be to observe that the integration required by simulation composability remains an efficient means to build a meaningful simulation.

3. Component-Based Software Design

We begin our discussion on CBSD with two definitions for a software component from widely read texts. The first is by Szyperski [11]:

A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.

Contrast this definition with [12]:

A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

At face value, these definitions may appear to be very different, though upon further inspection they are quite similar. Both definitions stipulate that components are independent, and can be composed with other components. The first definition is more specific about the interfaces and dependencies, though these characteristics are wrapped up within the conformance to a component model of the second definition. Clearly, the major difference between the definitions is that the second calls for composition without modification. If deployed components are not to be modified, then there is a major difference between the expectations of software engineers and simulationists with respect to the ability to modify components and still have composition. We expound on this point in section 4.

Since the inception of CBSD techniques in the early 1990s, the technology has quickly reached a level of maturity where its use is accepted across the software development arena today. Three component models have clearly risen to the top: Microsoft's Component Object Model Plus (COM+) [13], the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [14], and Sun's Enterprise JavaBeans (EJB) [15].²

While each of these models enforces a unique binary structure, in an abstract sense they are very similar in how they function. For example, to invoke a component service, the client process calls a local

² COM+ is closely related to ActiveX, OLE, COM, DCOM, and MTS. EJB is closely related to JavaBeans. For simplicity we have generalized the related technologies into one component model.

proxy, or object request broker, which then marshals the function parameters and ships them to the component. The component can reside within the same process, or in another process on the local machine or a remote machine. Regardless of the location of the component, the server where it resides will contain a remote proxy for the component, which then unmarshals the parameters and passes them to the component. The component then computes, and returns results through the proxies back to the calling client (which could itself be a component).

All three of these component technologies provide a means for components to expose their public interfaces, either through reflection, stubs, or a combination of both. Additionally, there is support in all three for dynamic invocation and the ability to query component features from information provided by metadata. In addition to these common services, there are also unique services provided by some subsets of the models, such as support for security, transactions, events, and serialization.

The aforementioned unique binary structure of these technologies is an obvious constraint on the user, and also limits the ability to build systems across component models. The technology to allow a component to use another component that is implemented in another component model is only available for certain pairs of component models. However, recent work has focused on providing a framework that supports interoperability across all component models [16].

In general, state of the art software component technologies solve syntactic composability challenges. They provide the implementation facilities for components to communicate and provide services amongst one another, but they provide no guarantees about the reliability or consistency of the exchanged information with respect to the rest of the computing environment. Leaders in the software engineering community admitted in 1998 that the current state of CBSD was challenged with respect to developing large applications, managing multiple versions of components, and integrating components developed by different people using different component frameworks [17]. For these reasons and others, industry survey results published in 2002 show that CBSD has not yet become the mainstream, predominant software development technology many envisioned it to be [18].

The discovery by component users that often leads to frustration is what software engineers call “side-effects,” or unintended consequences of adding or

swapping a new component into a system. One of the questions we explore next is whether these side-effects are fundamentally the same challenges faced by simulation developers who come across issues of semantic composability.

4. Closing the Perceived Gap Between CBSD and Composability

In their recent monograph on simulation composability, Davis and Anderson make the case that composing models is more difficult than composing general software components [19]. Their argument is predicated on the assumptions that models are more complex, are developed for particular purposes, and depend on context-sensitive assumptions. They believe model (simulation) composition infers a white-box where the developer can see, and quite possibly modify, the internals, and software engineering composition implies a black-box component with only the interface exposed.

When reviewing the historical progress of software engineers and simulationists, this argument seems compelling. However, in this section, we will examine this argument in more detail and show that this is not an accurate characterization; that indeed, software engineering and simulation composability challenges are fundamentally the same. In particular, the assumption that software engineers can disregard internal states that simulationists must analyze for meaning does not clearly hold. We compare composability and CBSD along four dimensions: the business case, architectural mismatch, the complexity of the composition, and component semantics.

4.1 The Business Case

As we outlined in section 1, the motivations for composition are consistent across M&S and software engineering. Both communities desire reuse and a wider range of design alternatives.

Throughout the literature, simulationists and software engineers also clearly agree on the business case challenges of composition. In both fields, there is an outcry for the infrastructure necessary to facilitate the motivation to build and use components. This infrastructure includes, in addition to the components themselves, repository facilities, meta-data standards, and query tools. The challenge is the large up-front investment necessary to jump-start the communities to make composition an every-day reality.

Fred Brooks, a pioneer in the field of software engineering, believes that, although not a “silver bullet”, component-based engineering is the key technology for future advances in software development [20][21]. However, he eloquently states why component reuse isn’t happening in the large. Simply put, the cost to reconstruct a component is low, and the cost to discover the functionality of an existing component is high.

Here is another perspective on the same challenge. If a conceptual component is small and simple, a developer will deem it easier to build and integrate it himself rather than trying to understand and integrate an existing component. However, if a conceptual component is large and complex, then the investment associated with componentizing the software for future reuse is too high to justify the lack of a short-term benefit, and could even cause short-term pain if a deadline is missed, notwithstanding the potential complexities associated with composing a piece of complex software. These arguments apply equally to both the simulation and software engineering fields, and hence at a business case and motivation level, simulationists and software engineers face the same challenges to gain some momentum for composability.

Brooks uses the field of mathematics to provide a good example where components have been embraced, and this example provides a useful context for simulationists [21]. In mathematics, it takes a large amount of intellectual effort to write software, so the cost to reconstruct a mathematical component is high. There is also a rich and standard nomenclature within the mathematical field to describe the functionality of components, so it doesn’t take long for a developer composing a mathematical component to understand the functionality and assumptions underlying the component. We believe this successful exemplar from a subset of the general software domain provides some hints for future successes of simulationists advocating composability. Composability will be most successful where it produces a significant reduction in development effort and provides formal means to describe the functionality of components. These ideas are not new to simulationists. However, some simulationists may be surprised to learn that they are not new to software engineers either.

4.2 Architectural Mismatches

A survey of some classic software engineering literature on CBSD provides a good starting point for providing comparisons between simulation composability and CBSD. In 1995 Garlan, et al.

explored the problems associated with building a useable system from large, monolithic components [22]. In addition to challenges of performance and code size, they realized a primary complication was caused by components that possessed incompatible assumptions (about program control, data, communications, topology, build process...). The solution required a significant rewrite of much of the existing component software to align component assumptions and make the pieces work meaningfully together. They coined the term “architectural mismatch” to describe how the components did not fit together well. Their recommendations for the way forward include the explicit announcement of assumptions, using orthogonal subcomponents, developing techniques for bridging mismatches, and providing a cookbook for software composition rules and principles.

In 1996 Sullivan and Knight attacked the premise of Garlan that building systems from large scale components is hard [23]. They built a system from large-scale components (each on the order of a million lines of code) in approximately eight man-days. Sullivan and Knight state that they did not experience the same challenges because they used components that were designed to work together from the start (Visio and Access, which both implemented the OLE component framework). Quoting their conclusion, “*A key lesson is that, if components are to be composable, they have to be designed for it.*” This is another example where simulationists and software engineers are coming to the same conclusion, as evidenced by stunningly similar statements to this one that are found in highly regarded publications on simulation composability (Davis and Anderson [19] and Kasputis and Ng [24]).

4.3 Scale Matters

While Sullivan and Knight are correct in asserting that building to a common framework is critical to composability, we believe it is also important to evaluate the complexity of the composition as opposed to the complexity of the components themselves. In Sullivan’s system, the main task of the composition consisted of maintaining consistency between the database representation of a fault-tree in one component and the graphical representation in another. Even though each component is approximately one million lines of code, this composition is relatively simple in comparison with the composition complexity of many modern simulation federations. We therefore use this work as an example where composition shows

promise, but additional research of complex compositions is required.

The tendency to keep composition relatively simple is a distinguishing trend we see in the software engineering community. The classic example where components have been widely and successfully used is with the construction of graphical user interfaces (GUIs). In these systems, the components are typically buttons or other widgets with very constrained capabilities. The dependencies between components are obvious to the developer and can be maintained without a lot of bookkeeping or conceptual effort. The semantics of the components are so constrained that it allows for easy reuse when the widget is used in a different context. For example, the typical meaning (semantics) for a button is that it triggers an event when it is clicked or released. There are no underlying assumptions or dependencies that are not obvious from the context. There could possibly be added complexity such as the color of the button dynamically changing. However, the key point here is that there are only so many interesting things one can do with a button, and hence button components are easy to compose within the domain for which they were intended, a GUI.

As Davis pointed out, models are complex, and hence very difficult to compose. We argue that this point is not necessarily a truth, but rather an artifact of the way simulationists have attacked the composability problem. For instance, there are a multitude of legacy monolithic simulations existing in the world today. Simulationists tend to treat these simulations as components with which to build new simulations. Any attempt to compose these very complex components will undoubtedly result in the kind of architectural mismatches described by Garlan, mismatches that can be described as both syntactic and semantic composability challenges.

On the other hand, software engineers have achieved successes in CBSD by building smaller components, engineering them to a common framework, and then composing them within a common domain (GUIs, for example, are one well-known domain) where it is easier to manage the assumptions and dependencies. We believe simulationists should consider these successes in software engineering and start to build success stories by composing smaller pieces that are designed to work together, and then scaling up. OneSAF appears to be a good example where this methodology can be applied.

Computer simulations are a subset of all computer software, and simulation components are a subset of all

software components. Hence, it follows that the simulation composability problem cannot be any more difficult or complex than the general software composability problem. We argue that simulationists encounter complex composition challenges because they are working on problems with a larger scale. However, simulationists do have a unique perspective of the challenges associated with trying to compose components that have a large number of complex dependencies and assumptions. These are the semantic challenges with which software engineers are now beginning to come to terms.

4.4 Semantics Matter

As referenced earlier, the argument exists that simulationists need to treat components as white boxes with a license to see, and even modify if necessary, the internals. And the same argument states that software engineers treat components as black boxes, assuming the correctness of the internals and caring only about the interfaces. Another way to characterize this argument is that software engineers are concerned with syntactic composability, and simulationists are focused on semantic composability. In other words, general software components are just that, software, without meaning outside of the software context. This is another example where the differences in the complexity of the composability problem that each community has historically attempted to solve has influenced our beliefs.

We believe semantic composability is very important to software engineers. Because semantics are defined by component internals, software engineers are very much in tune with component internals. Clearly, if syntax were all that mattered to software engineers, they would not write in their publications “*It is hard to detect errors when components are used in a different context than for what they were originally designed.*” [25] Rather, it has historically been the case that the semantics of typical software engineering components (GUI widgets for example) and the limited domains in which the components were used have been simple enough that solutions for syntactic composability (CBSD) were also solutions for semantic composability.

Another perspective on whether CBSD components have semantics is one in which we consider GUI widgets as models of real world entities. For example, a button widget is a model of any physical button that triggers a mechanical or electrical event. The semantics of a button widget are obviously important to the engineer employing it. For example, if button-

widget *A* were semantically defined to disappear each time it was clicked, this would be very important to the GUI developer who tried to compose budget-widget *A* with his interface. If he did not want this button to ever disappear, then the composition of this button with the other components could result in an interface that was not meaningful.

As an aside, and not germane to the central arguments in this paper, one could even take this point one step further and show that all software is just a model of some computation being performed on a mechanical Turing Machine. Hence, all software is a model, and software composability is equal to model composability. This conjecture blurs the lines between syntactic and semantic composability, since if all software is a model, then it all has meaning, and hence all syntax is semantics. Without delving further into this topic, we believe that the traditional delineation between syntactic and semantic composability, while very convenient, does not necessarily have a clearly defined boundary.

Software engineers care as much about semantics as simulationists. As the complexity of composition in general software components is scaled up, software engineers are discovering that semantics matters. And this scaling up is starting to happen, as evidenced by this journal quote: *“The goal is to replace IDE palettes of text fields, data grids, push-buttons, and similar GUI controls with palettes that contain business objects, services, and functional views. Developers would then select, customize, and assemble these items into specialized components such as insurance coverage, and script complex business processes such as order fulfillment.”* [26] As software engineers increase the complexity of their component interactions, we believe they will (or probably already have) come across the same semantic composability challenges already discovered by simulationists. There are technologies and standards being developed today outside the simulation community to address this challenge. They are discussed in section 5.

4.5 Convergence in CBSD and Composability Research

Simulationists and software engineers working on composability have converged upon the same results, but from different directions. Simulationists started by trying to compose large, monolithic components. Software engineers achieved successes by composing small, semantically trivial components.

From different directions, syntactic composability has been achieved in both communities. DIS, ALSP, and the HLA provide the glue to tie together simulation components, while COM+, EJB, and CORBA provide binary compatibility for the interoperability of software components. Clearly, the great unsolved challenge that lies ahead for both communities is semantic composability, the capability to compose components, with the composition resulting in a meaningful, valid system.

5. Moving Ahead with Simulation Composability

Current simulation interoperability advances have benefited from past software engineering research. Specifically, the HLA design is heavily influenced by the principles of object-oriented analysis and design, to include information hiding and hierarchical class structures. Not only has past software engineering research impacted heavily on the current state of simulation interoperability, but it will have future impacts also. Observing how simulation composability and CBSD are fundamentally the same, have experienced similar progress, and have some common challenges ahead, it follows that the simulation community needs to closely monitor the new standards and technologies emerging from the software engineering community as potential paths for progress.

5.1 Following this Path

We will highlight three software technologies as potential avenues for simulation advances in composability. We chose technologies that provide potential for the simulationists to document a simulation and formally reason over the documentation to provide insights about the prospects for composability. This is not intended to be an exhaustive list.

5.1.1 Predictable Assembly from Certifiable Components (PACC) and Prediction-enabled Component Technology (PECT)

Software engineers have discovered that even though technologies exist to plug together compiled components (CORBA, EJB, COM+), only after assembly and testing might it be discovered that these components are incompatible. Their interfaces may not be sufficiently descriptive and the behaviors of individual components may be unknown. These shortcomings prevent the discovery of the behaviors of compositions until after assembly.

PACC is an initiative at the Software Engineering Institute to predict and certify the runtime behavior of an assembly of components [27]. PECT is an approach to achieving the PACC objectives [28]. PECT is an extended component model that works by building a Reasoning Framework for any runtime property. The Reasoning Framework consists of Property Theory (logic for reasoning about the property) and Decision Procedure (an automated means of computing predicted properties). The Reasoning Framework must explicitly expose any assumptions about the system it models. The component technology then ensures that components and assemblies satisfy these assumptions through static checking, resulting in a predictable assembly behavior. In addition to this static check, each well-formed assembly can be specified with a composition language, which is formally mapped to a unique model in the reasoning framework, introducing more constraints on the components and their assemblies.

This technology provides potential opportunities for documenting and reasoning about the semantics of simulation components and compositions of simulations in a formal way. We believe this formal reasoning is exactly the sort of technology needed to support simulation semantic composability.

5.1.2 Web Ontology Language (OWL)

The Semantic Web is the vision of Tim Berners-Lee, the founder of the World Wide Web. His vision is to "... bring structure to the meaningful content of Web pages." [29] OWL is an emerging, enabling technology for the Semantic Web under the guidance of the World Wide Web Consortium [30]. Its power lies in the ability to define structured ontologies for delivering richer integration and interoperability of data among descriptive communities. In OWL, data is described using formal terms including discrete math concepts (e.g. cardinality, enumeration, relations, and set logic) and class hierarchies. Ontologies are built using OWL to describe the structure and meaning of data specific to any domain. OWL also includes formalisms to tie together ontologies, providing consistent meaning of information across domains.

OWL is a definite candidate technology for documenting simulation components, and work has already begun in this endeavor [31]. Using OWL formalisms, pre-built ontologies supporting any domain for which a model description is needed (to include the M&S domain), and tools to reason about ontologies, a framework is possible with which to

support the reasoning about whether simulation components are semantically compatible.

5.1.3 Unified Modeling Language (UML) and Model Driven Architecture (MDA)

UML is an OMG standard providing a graphical tool for modeling the structure, behavior, and management of software applications [32]. UML has become very widely used in industry for software development and documentation tasks. MDA, also an OMG product, provides the means to separate application logic from platform technology [33]. Using the MDA, a developer builds a Platform Independent Model (PIM) in UML, describing the application while abstracting any potential platform technology. Then, using standardized mappings and supported by automated tools, the PIM can be transformed to a Platform Specific Model (PSM), also described in UML. This automated transformation allows for the rapid implementation of a PIM to any one of many supported technologies, to include CORBA, EJB, and .NET. The MDA also provides facilities for building repositories of models, describing data structure, and transmitting models.

Previously, simulationists have proposed using the OMG's MDA to support simulation interoperability and reuse [34][35][36]. These ideas, if successful, would lead to a much more sophisticated and automated means of providing HLA-like services to simulations built from components. While useful, these approaches don't attack what we believe to be the critical need for the simulation community, some level of semantic composability. However, Ng et al. does acknowledge the need for a separate XML-based model specification language to support semantic composition [35].

Davis and Anderson embrace documentation as a critical supporting element of composability [19]. They advocate the potential of a high-level graphical specification system like UML coupled with a more detailed specification system, for example the Discrete Event Specification System (DEVS) [37], as a viable approach for documentation. If properly used to document component functionality, constraints, and assumptions, and given the right tools to query this documentation, a more automated approach for achieving semantic composability is possible.

5.2 Leveraging the Uniqueness of Simulations

Guaranteeing semantic composability is a very hard problem. The previous section outlined technologies

that may provide approaches to this problem by exposing the internals of components. However, success is not guaranteed, and will probably be achieved at a slow pace. Therefore, we believe multiple approaches are in order.

In Section 4.3 we argued that the composability of simulation components is fundamentally no more difficult than the composability of general software components. However, because simulation composability is a subset of software composability, focusing on unique solutions for simulation composability could potentially provide solutions that are not necessarily applicable for all software. Hence, simulation composability could be a less difficult problem than software composability. We are investigating properties and characteristics commonly found in simulation software, not normally found in all software, from which we can gain traction and leverage for unique simulation composability solutions.

For example, event-based software is an active area of research within the software engineering community [38]. However, the software engineer's perspective in event-based software is clearly on event handling. We see significant opportunity in separating out event generation in any model we consider for large-scale simulations. Event generation in typical simulations includes assumptions about phenomena that are simulated, and those that are not, and the use of stochastics as a substitute for the unknown and/or unsimulated phenomena.

Stochastic sampling is another property commonly found in simulations. The presence of stochastics implies a relaxed level of strictness about the values that can be taken on by a parameter at any given point in time. We want to leverage this relaxation of parameter values in the determination of the suitability of two components for semantic consistency.

Lastly, simulations typically have some form of time management. Regardless of whether a simulation is event-stepped, or time-stepped, the simulation, by the very nature of it running on a computer, has a discrete flavor to it. Typically, there are places along the discrete time axis where things need to be made semantically consistent, and in between these times, this condition can be relaxed. In other words, semantic consistency is not a continuum that follows a continuous curve along the axis of time. Once again, we want to leverage this identification of critical points and relaxation of consistency in determining whether components are semantically composable.

6. Conclusion

We have shown that the CBSD and simulation composability are more similar than different. They both have the goal of promoting reuse and widening design alternatives. Both are challenged with respect to putting together the right business case to get the technologies jump-started. Fundamentally, both software engineering and simulation composition are predicated on syntactic and semantic composability, which means that both the interfaces and the internals are necessary to make composition work. At this point in their maturity, both simulation composability and CBSD have converged. They both have made significant progress solving technical interoperability problems, yet both are challenged ensuring that compositions are meaningful and valid.

We believe a two-pronged approach provides the best possibility for future advances in simulation composability. First, because composability and CBSD share the same challenges, simulationists must continue to monitor the progress of software engineering research for potential technologies to provide further advances in interoperability and composability. Secondly, because simulation components have unique characteristics not necessarily present in all software components, attempts should be made to leverage these properties to make advances that might be applicable only within the simulation domain.

7. Acknowledgements

We wish to acknowledge support from the Defense Modeling and Simulation Office, particularly from Sue Numerich and Phil Zimmerman. Additional support was provided by the U.S. Army and the National Science Foundation (ITR 0426971).

8. References

- [1] Mikel D. Petty and Eric W. Weisel. "A Composability Lexicon." *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, Orlando, FL, April 2003.
- [2] Ernest H. Page and Jeffrey M. Opper. "Observations On the Complexity of Composable Simulation." *Proceedings of the 1999 Winter Simulation Conference*, Phoenix, AZ, December 1999.
- [3] Mikel D. Petty, Eric W. Weisel, and Roland R. Mielke. "Computational Complexity of Selecting Components For Composition." *Proceedings of*

- the Fall 2003 Simulation Interoperability Workshop*, Orlando, FL, September 2003.
- [4] Michael Roy Fox, David C. Brogan, and Paul F. Reynolds, Jr. "Approximating Component Selection." To appear in *Proceedings of the 2004 Winter Simulation Conference*, Washington, D.C., December 2004.
- [5] Mikel D. Petty and Eric W. Weisel. "A Formal Basis For a Theory of Semantic Composability." *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, Orlando, FL, April 2003.
- [6] Eric W. Weisel, Mikel D. Petty, and Roland R. Mielke. "Validity of Models and Classes of Models in Semantic Composability." *Proceedings of the Fall 2003 Simulation Interoperability Workshop*, Orlando, FL, September 2003.
- [7] "IEEE Standard for Distributed Interactive Simulation – Application Protocols." IEEE std. 1278.1
- [8] Richard Weatherly, David Seidel, and Jon Weissman. "Aggregate Level Simulation Protocol." *Proceedings of the 1991 Summer Computer Simulation Conference*, Baltimore, MD, July 1991.
- [9] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)". IEEE std. 1516-2000.
- [10] Robert L. Wittman, Jr. and Cynthia T. Harrison. "OneSAF: A Product Line Approach to Simulation Development." Technical Report, The MITRE Corporation, February 2001.
- [11] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2d Edition)*. Addison-Wesley, 2002.
- [12] William T. Councill and George T. Heineman. "Definition of a Software Component and its Elements", chapter in *Component Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
- [13] Component Object Model Plus (COM+) homepage. Microsoft, <http://www.microsoft.com/com/tech/COMPlus.asp>
- [14] Common Object Request Broker Architecture (CORBA) homepage. Object Management Group, <http://www.omg.org/corba/>.
- [15] Enterprise JavaBeans Technology (EJB) homepage. Sun Microsystems, <http://java.sun.com/products/ejb/>.
- [16] Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri. "The Vienna Component Framework Enabling Composition Across Component Models." *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.
- [17] Alan W. Brown and Kurt C. Wallnau. "The Current State of CBSE." *IEEE Software*, 15(5):37-46, September-October 1998.
- [18] Paul Allen. "CBD Survey: The State of the Practice." *The Cutter Edge* email service, Cutter Consortium, April 2002.
- [19] Paul K. Davis and Robert H. Anderson. *Improving the Composability of Department of Defense Models and Simulation*. RAND National Defense Research Institute, Santa Monica, CA, 2003.
- [20] Frederick P. Brooks, Jr. "No Silver Bullet." chapter in *The Mythical Man-Month (Anniversary Edition)*, Addison-Wesley, 1995.
- [21] Frederick P. Brooks, Jr. "No Silver Bullet Refired." chapter in *The Mythical Man-Month (Anniversary Edition)*, Addison-Wesley, 1995.
- [22] David Garlan, Robert Allen, and John Ockerbloom. "Architectural Mismatch or Why It's Hard to Build Systems Out of Existing Parts." *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle, WA, April 1995.
- [23] Kevin J. Sullivan and John C. Knight. "Experience Assessing an Architectural Approach to Large-Scale Systematic Reuse." *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996.
- [24] Stephen Kasputis and Henry C. Ng. "Composable Simulations." *Proceedings of the 2000 Winter Simulation Conference*, Orlando, FL, December 2000.
- [25] Kelli Houston and Davyd Norris. "Software Components and the UML." chapter in *Component Based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
- [26] David Krieger and Richard M. Adler. "The Emergence of Distributed Component Platforms." *IEEE Computer*, 31(3):43-53, March 1998.
- [27] Predictable Assembly From Certifiable Components (PACC) homepage, Carnegie Mellon University/Software Engineering Institute, <http://www.sei.cmu.edu/pacc>.
- [28] Kurt C. Wallnau. *Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC)*, Technical Report, Carnegie Mellon University/Software Engineering Institute, April 2003.
- [29] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web." *Scientific American*, 284(5):34-43, May 2001.
- [30] Web Ontology Language (OWL) homepage. World Wide Web Consortium, <http://www.w3.org/2003/OWL/>.

- [31] John A. Miller, Gregory T. Baramidze, Amit P. Sheth, and Paul A. Fishwick. "Investigating Ontologies for Simulation Modeling." Proceedings of the 37th Annual Simulation Symposium, Arlington, VA, April 2004.
- [32] Unified Modeling Language (UML) homepage. Object Management Group, <http://www.omg.org/uml/>.
- [33] Model Driven Architecture (MDA) homepage. Object Management Group, <http://www.omg.org/mda/>.
- [34] Andreas Tolk. "Avoiding Another Green Elephant – A Proposal For the Next Generation HLA Based On the Model Driven Architecture." *Proceedings of the 2002 Fall Simulation Interoperability Workshop*, Orlando, FL, September 2002.
- [35] Wei Tze Ng, Seng Joo Thio, and Cheng Hong Teo. "A MDA-Based Translation Approach to Component-Level Reuse." *Proceedings of the Spring 2004 Simulation Interoperability Workshop*, Arlington, VA, April 2004.
- [36] Don Brutzman and Andreas Tolk. "JSB Composability and Web Services Interoperability Via Extensible Modeling & Simulation Framework (XMSF), Model Driven Architecture (MDA), Component Repositories, and Web-based Visualization." Technical Report, Naval Postgraduate School and Old Dominion University, November 2003.
- [37] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems (2d Edition)*. Academic Press, 2000.
- [38] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. "A Framework For Event-Based Software Integration". *ACM Transactions on Software Engineering and Methodology*, 5(4):378–421, October 1996.

Author Biographies

ROBERT G. BARTHOLET is a Ph.D. Candidate in Computer Science and a member of MaSTRI at the University of Virginia. Robert earned his B.S. in Electrical Engineering at the U.S. Military Academy at West Point, and his Masters in Computer Science at the University of Virginia. Robert is an active duty Lieutenant Colonel in the U.S. Army with 9 years experience in the Field Artillery. For the past 7 years he has served in the Signal Corps as an Information Systems Engineer supporting battlefield communication and automation systems in tactical units.

DAVID BROGAN earned his PhD from Georgia Tech and is currently an Assistant Professor of Computer Science and a member of MaSTRI at the University of Virginia. For more than a decade, he has studied simulation, control, and computer graphics for the purpose of creating immersive environments, training simulators, and engineering tools. His research interests extend to artificial intelligence, optimization, and physical simulation.

PAUL F. REYNOLDS, Jr. is a Professor of Computer Science and a member of MaSTRI at the University of Virginia. He has conducted research in modeling and simulation for over 25 years, and has published on a variety of M&S topics including parallel and distributed simulation, multiresolution models and coercible simulations. He has advised numerous industrial and government agencies on matters relating to modeling and simulation. He is a plank holder in the DoD High Level Architecture.

JOSEPH C. CARNAHAN is a Ph.D. Candidate in Computer Science and a member of MaSTRI at the University of Virginia. Joseph earned his B.S. in Computer Science at the College of William and Mary, and has held the position of Scientist at the Naval Surface Warfare Center, Dahlgren Division.