On Validation of Semantic Composability in Data-driven Simulation *

Claudia Szabo and Yong Meng Teo
Department of Computer Science
National University of Singapore
Computing 1, 13 Computing Drive
Singapore 117417
Email: claudias, teoym@comp.nus.edu.sg

Abstract

A simulation model composed using reusable components is semantically valid if it produces meaningful results in terms of expressed behaviors and meets the desired objective. This paper focuses on the validation of data-driven component-based modeling and simulation. In data-driven simulation applications, it is necessary to model entity behavior at higher resolution. In simulations such as military training scenarios where entity behavior changes dynamically, additional input data is required to express complex state transitions. This can significantly increase the composed model state space and presents a major challenge in simulation validation. Using a component-based data-driven tactical military simulation, we propose a layered and automated approach for semantic composability validation. While the expressivity of data-driven models increases the semantic equivalence of the validated model, it incurs higher validation cost.

1. Introduction

Component-based simulation model development is an appealing approach to the simulation community [14] because it reduces the time and cost of developing complex simulations. Simulation composability [19] can be defined as "the capability to select and assemble simulation components in various combinations to satisfy user requirements". Component-based frameworks that employ reused simulation components promise shorter development time and increased flexibility in meeting diverse user needs [19]. In modeling and simulation, two main levels of composability have been identified, namely syntactic composability and semantic composability [5]. In syntactic composability, components have to be properly connected and must interoperate, which assumes common communication protocols, data formats, as well as a common understanding of the time management mechanisms employed. In semantic composability, the composition must be meaningful for all components involved. Furthermore, the composed model must be valid [19]. This is because simulation models are widely used to make critical decisions and to answer "what-if" questions [3]. For example, under the current US Department of Defense policy, all models and simulations must undergo a costly verification, validation, and accreditation (VV&A) process [29]. As such, semantically valid simulation models are absolutely necessary [26], and thus component-based simulation frameworks must provide for the validation of semantic composability or at least for the context in which semantic composability can be achieved [14].

*A version of this paper is published in the Proceedings of 24th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation, pp. 73-80, IEEE Computer Society Press, Atlanta, USA, May 17-19, 2010.

In software engineering, the validation of the composed software artifact focuses on the overall program correctness and on ensuring that component methods are executed in the correct order according to some protocols [15]. In contrast, a valid simulation model is one that mimics closely the real system that the simulation model abstracts [2]. Here, while overall program correctness is required, it is very important for the simulation to behave exactly (or close to) 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 [9, 26]. Thus, the simulation model validation process is often manual, lengthy, and requires the presence of a system expert [2, 3]. Similarly, 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 [13, 17]. For example, the process of Verification, Validation and Accreditation (VV&A) for modeling and simulation in the US Department of the Navy defines seven user roles and thirteen important steps grouped in five categories, namely conceptual model validation, design verification, implementation verification, and results validation [10]. The difficulty of military simulation model validation process is exacerbated by the complexity of the models. For example, in military training simulations, the state of a 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). 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 [9].

The validation of composable simulations is a non-trivial problem [3, 9, 19, 26]. 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 [2]. Next, reused components are developed for different purposes and when composed may result in emergent properties [12]. Similarly, the context in which a reused component was developed and validated might differ from the new context of the composed model [5, 26]. 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" [14]. The motivation of our work is twofold. Firstly, simulation model validation is a lengthy, manual process that can possibly be improved if a component-based paradigm is applied. Secondly, well-established software verification techniques 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 [30], Petty and Weisel's theory of composability [19], and component abstractions such as BOM [18].

We consider a simulation 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. Constraint validation [25] is the process of verifying the communication of connected components for semantic correctness. It includes validating that messages passed between components are syntactically correct, and semantically meaningful with respect to a communication XML schema and a component-based ontology [25]. In our previous work, we proposed a layered approach to the automated semantic validation of compositions in simulation model integration with increasing accuracy and complexity [23, 24]. Our approach focuses on the validation of general model properties such as semantic correctness of component communication, safety and liveness of the logical component coordination in the context of instantaneous transitions and over time, as well as on providing a formal guarantee of the composition validity. Our layered validation process is implemented in CoDES, a component-based modeling and simulation framework that facilitates component reuse, hierarchical composition of components within and across application domains, syntactic composability verification, and semantic composability validation [22].

In this paper, we discuss the application of our proposed validation process to the complex field of mil-

itary training simulations, in which base components are data-driven entities and compositions are complex systems with large simulation state space. We present the Military Training application domain as it is added to the XXX framework. Next, we show how a military training simulation of a tank versus a soldier troop attack is validated. The validation of data-driven composed models is an improvement of our previous work where the component state machine was less complex and components were not data-driven. In our experiment, we validate a tank versus a soldier troop military training simulation. The contributions of this paper include:

1. An approach for validation of data-driven simulation composed from complex components whose actions depend on the input data received. The application is a closed system with feedback loop. Our validation process guarantees overall *model correctness* from a software engineering perspective, as well as *model validity* from a simulation perspective.

2. The application of our validation process in the validation of military training simulations. Successful application of our validation process has the potential to greatly improve the lengthy military verification, validation, and accreditation process, in the military training application domain.

3. Valuable insight into the problems that arise when a manual process such as simulation validation is automated. Complex models and components incur abstraction trade-offs and the notion of simulation validity defined as conforming to the real system is difficult to capture formally. Furthermore, the complex components result in an increased validation runtime.

This paper is organized as follows. Section 2 describes how composable data-driven simulations are extended in our component-based simulation framework. Section 3 shows how a military training simulation of a tank versus soldier troop attack is validated. We discuss related work in Section 4 and present our concluding remarks and future work in Section 5.

2. Composable Data-driven Simulation

2.1. Framework Overview

CoDES is a hierarchical component-based framework for modeling and simulation [22] in which a simulation component is modeled as a meta-component, an abstraction of the actual component implementation. The meta-component describes the *attributes* and *behavior* of a component and is used to support model discovery, and syntactic and semantic validation. 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. More formally, a component C_i is represented by the tuple $C_i = \langle R, A_i, B_i \rangle$, where R denotes mandatory attributes that are common to all components, A_i denotes component specific attributes, and B_i represents component behavior as a state machine. A component behavior is represented as a timed automata as follows:

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

where I_l is the set of input data; S_p is the current state; Δt is the transition duration; $Cond_m$ defines the condition(s) for the state transition; S_t is the next state; O_l is the set of outputs after the state change; A_m is the set of modified attributes after the state change.

Reusable components are divided into three categories in the model repository. *Base components* are well-defined atomic entities specific to an application domain. For example, in Queueing Networks, the base components could be a *Source* that generates jobs, a *Server* that services jobs, and a *Sink* that displays job data. On the other hand, for the Military Training application domain, two base components could be *Tank* modeling an army tank, and a *SoldierTroop*, modeling a troop of soldiers. The separation into different application domains helps to capture application domain knowledge that can be otherwise difficult to express. This enhanced domain knowledge is employed in syntactic composability verification, component discovery, and semantic composability validation.

A developed simulation model can be reused as a *standalone simulator* or as *model components* in a larger simulation model. Model components can also be reused across application domains. This ensures that the framework has both *breadth* in covering many application domains, and *depth* in covering an application domain in detail. In the adopted component-connector paradigm, components are black-boxes linked by connectors such as fork and join to support message passing. Composition grammars determine the syntactic composability of simulation components [22]. COSMO, our component-oriented ontology and COML, our proposed component markup language, facilitate component discovery and semantic validation of compositions [25].

2.2. Military Simulation

To add a new application domain involves extending our component-oriented ontology by providing descriptions of the domain's base components, including defining attribute and property hierarchies. Next, the framework composition grammar is extended by adding composition rules specific to the new application domain.

Tactical Military Training Simulation

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. As in Figure 1, a new simulation model is developed using our graphical input model interface by drag-and-drop icons representing soldiers and tank. The conceptual model is a closed system with feedback loop. Next, the conceptual

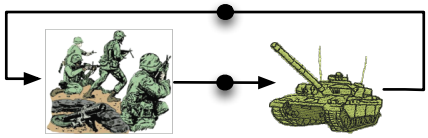

Figure 1. Tank vs Soldier Troop Training Simulation

model is syntactically verified against the new extended composition grammar. If the conceptual model is syntactically correct, each base component can be individually discovered based on attributes and behavior information provided by the user. The discovery service ranks relevant components based on semantic knowledge stored in our component-oriented ontology.

2.3. Data-driven Components

Assume that the best candidates returned by the discovery service for the Tank and SoldierTroop base components are components $tank_1$ and $troop_1$ respectively. Table 1 presents the most important parts of the component state machines. The combined state machines for the two components is shown informally in Figure 2, 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 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 [28]. 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. As it can be seen, $tank_1$ and $troop_1$ are data-driven base components.

Entity	Attribute	Input	Output	State Machine
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*
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 1. Meta-component Information

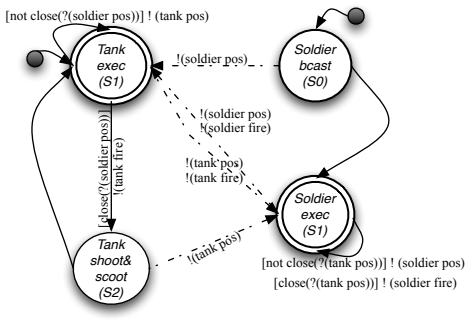

Figure 2. Data-driven Component Interaction

The new version of COML caters for data-driven components in two ways. Firstly, attribute names and values can now 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 1 could contain only simple logic such as the one from C_3. In the new COML version, conditions, e.g. $C_1, \ldots, C_4$, as well as attribute change sections, e.g. $A_1, \ldots, A_4$, can contain complex logic based on specific input data attribute values, such as the opponent's positions in Figure 3. 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. We have modified the inital COML specification to cater for data-driven components as follows. Using the new COML version, we show in Figure 3 that $tank_1$ expects an auxiliary attribute with two fields in its input message. Next, 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                          </data>
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>
```

Figure 3. Data-driven Component Representation

is now parsed for state changing conditions and attribute modifications, such as the one for C_1.

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 [25]. By *type* we mean either a base component type such as Tank or SoldierTroop, or a general *ModelComponent* type which 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 truth value of condition C_1. Similar structure is found in the attribute values modification section in our COML schema. In the new version of COML, Java code mixed is with XML tags to facilitate backward compatibility with the existing components in the framework repository. Currently, in the CoDES repository there are approximately two thousand Queueing Networks models [22].

3. Semantic Composability Validation

In contrast to military training simulation validated using a lengthy and manual process involving simulation experts [17], we show how validation is automated in our component-based simulation framework. Figure 4 presents the structure of our three-layer validation process organized in two steps. For a composed simulation model developed from reused components from the repository, we first validate desired model properties. In (1.1), we validate that component communication is semantically meaningful according to the COSMO ontology [25]. All subsequent layers assume that the component communication is meaningful and correct. In (1.2), the Concurrent Process Validation layer (CPV), we validate 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 the Meta-Simulation Validation layer (MSV), and validate safety, liveness and deadlock freedom using sampled time values for the time attributes. Contrary to the CPV layer where we employ formal definitions of safety and liveness, in this layer safety and liveness are defined from a practical perspective tailored to the modeling and simulation domain. Once the first level is complete, we have the guarantee that the composed model achieves safety, liveness, and deadlock free properties both formally with respect to component coordination, and practically considering simulation considerations such as time, attribute values provided by the user, and desired output. We can say that the first level validates the composition from a software engineering perspective. However, the composition may still be invalid from a simulation perspective, when compared to the real system. In the second level, we formally compare the model to a perfect model from the repository by analyzing how close the component execution is to the perfect model. We consider the perfect model to be the representation of the real system that the composed model simulates.

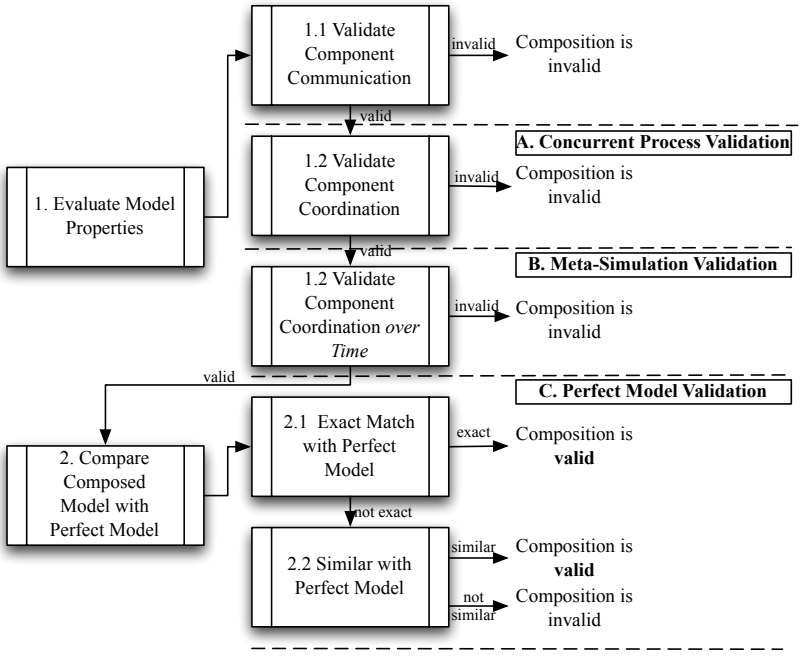

Figure 4. Layered Approach to Semantic Validation

3.1. 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 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 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 5 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 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 [6].

```
1  mtype {MSG}; chan to1 = [10] of {mtype}; ...
2  proctype CON_ONE_TO_ONE(chan in, out)
3  {do :: in ? MSG -> out ! MSG; od}

5  proctype TANK(byte id; chan in, out){
6  S1: atomic{ if :: in ? MSG -> if
7    :: out ! MSG -> progress: printf("MSG sent\n");
8                    goto S1; fi fi}}

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 -> if
17 :: 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); }
```

Figure 5. Simplified Tank vs Soldier Troop in Promela

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 [23]. 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 from a message passing perspective, the resulting Promela specification would be that presented in Figure 5. 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 6 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 6 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.

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

Figure 6. Detailed Tank vs Soldier Troop Attack in Promela

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 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 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 component. The value of the transient predicate is ideally provided by the component creator in the meta-component as shown in Table 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.

3.2. Perfect Model Validation

In step 2, a model M composed of $tank_1$ and $troop_1$ is validated by comparison with a perfect model M^* consisting of perfect components $tank^*$ and $troop^*$. A perfect 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 perfect 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 perfect 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 perfect component $troop^*$ to be the same as component $troop_1$ from Table 1. Let component $tank^*$ state machine be the one presented in Table 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
tank*	**Input** I_1, constraints: $class = PositionInfo$ $origin = SoldierTroop$ I_2, constraints: $class = InputFire$ $origin = SoldierTroop$ **Output** O_1, constraints: $class = PositionBroadcast$ $destination = SoldierTroop$ O_2, constraints: $class = OutputFire$ $destination = SoldierTroop$	$I_1S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_1$ $I_1S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_2$ $I_2S_1(\Delta movingTime) \xrightarrow{C_1} O_1S_1A_3$ $I_2S_1(\Delta shootingTime) \xrightarrow{C_2} O_2S_1A_4$ $I_2S_1 \xrightarrow{C_3} O_1S_1$ $I_1S_1 \xrightarrow{C_3} O_1S_1$ 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 2. Perfect 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 [21], and (v) *Bisimulation Validation* in which the execution of model M is validated against the execution of model M^* [24].

In Definition 1, 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. definitionDefinition

Definition 1. *(*Component) A simulation component, C_i, is defined as 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 intervals at which the component changes state.

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 3. f_2 is described first, because according to the state machine in Table 3, it is the $troop_1$ component that will initiate the communication.

	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 3. Formal Component Representation

Mathematical Composability

Next, the function composability is validated in the *Mathematical Composition* step. Following Definition 2, we obtain constraints for the values of var_1, var_2,var_3, var_4 and $var_{21}, var_{22}, var_{23}$ respectively.

Definition 2. *(*Mathematical Composability) Given a composed model $M = \{(f_i, f_j) | i \neq j, i, j = \overline{1, n}\}$, where f_i outputs and f_j requires input with time values $T_i^{out} = \{t_m^{(i)} | 1 \leq m \leq |O_i|\}$, and $T_j^{in} = \{t_n^{(j)} | 1 \leq n \leq |I_j|\}$, respectively. Then f_i and f_j are composable iif there exists the bijective binary relation $R = \{(t_n^{(j)}, t_m^{(i)}) \in T_j^{in} \times T_i^{out} | t_n^{(j)} > t_m^{(j)}\}$.

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:

$var_1 \geq 0, var_1 \geq var_2 1 + 50 + \Delta w_{12}$,
$var_2 \geq var_1 + 20, var_2 \geq var_{22} + 10 + \Delta w_{12}$,
$var_3 \geq var_2 + 40, var_3 \geq var_{22} + 13 + \Delta w_{12}$,
$var_4 \geq var_3 + 10, var_4 \geq var_{23} + 30 + \Delta w_{12}$.
$var_{21} \geq 0 + \Delta w_{21}$,

$var_{22} \geq var_{21} + 50, var_{22} \geq var_1 + 20 + \Delta w_{21}$,
$var_{23} \geq var_{22} + 13, var_{23} \geq var_2 + 40 + \Delta w_{21}$.

Next, the constraints are solved by the Choco constraint solver [7]. 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 perfect 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 perfect composition, in Figure 7(a) and Figure 7(b). Each interleaved execution is represented as a Labeled Transition System, $L(M)$ and

$f_2(\emptyset, s_1^2, 0) \rightarrow (O_1, s_2^2, 0)$
$f_1(I_1, s_1^1, 4) \rightarrow (O_1, s_2^1, 54)$
$f_2(I_1, s_2^2, 56) \rightarrow (O_1, s_3^2, 76)$
$f_1(I_1, s_2^1, 79) \rightarrow (O_2, s_3^1, 89)$
$f_1(\emptyset, s_4^1, 89) \rightarrow (O_1, s_5^1, 92)$
$f_2(I_2, s_3^2, 91) \rightarrow (O_1, s_4^2, 131)$
$f_2(I_1, s_4^2, 131) \rightarrow (O_2, s_5^2, 141)$
$f_1(I_1, s_4^1, 131) \rightarrow (O_2, s_5^1, 161)$
$f_1(\emptyset, s_5^1, 161) \rightarrow (O_1, s_6^1, 168)$
$f_2(I_2, s_3^2, 166) \rightarrow (O_1, s_6^2, 246)$

(a) Composition

$f_2^*(\emptyset, s_1^2, 0) \rightarrow (O_1, s_2^2, 0)$
$f_1^*(I_1, s_1^1, 4) \rightarrow (O_1, s_2^1, 54)$
$f_2^*(I_1, s_2^2, 56) \rightarrow (O_1, s_2^2, 76)$
$f_1^*(I_1, s_2^1, 79) \rightarrow (O_2, s_3^1, 89)$
$f_2^*(I_2, s_3^2, 91) \rightarrow (O_1, s_4^2, 131)$
$f_1^*(I_1, s_3^1, 134) \rightarrow (O_2, s_4^1, 164)$
$f_2^*(I_2, s_4^2, 166) \rightarrow (O_2, s_5^2, 176)$
$f_1^*(I_2, s_4^1, 179) \rightarrow (O_2, s_5^1, 199)$
$f_2^*(I_2, s_5^2, 201) \rightarrow (O_2, s_6^2, 281)$
$f_1^*(I_2, s_5^1, 284) \rightarrow (O_2, s_6^1, 314)$

(b) Perfect Composition

Figure 7. Interleaved Execution Schedules

$L(M^*)$ respectively, as shown in Figure 8. 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 [11].

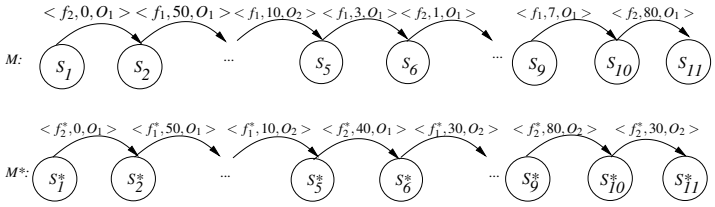

Figure 8. LTS Representation of Model Execution

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 $d(S_i, S_j^*) \leq \varepsilon$. 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 $\varepsilon = 0.25$ and obtain the following related nodes: $V_\varepsilon = \{(S_i, S_j^*) \| i \neq 5,6,9\}$, 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 $L_1(M^*)$. Figure 9 presents the resulting validation report detailing the executed layers and their respective results.

```
1  VALIDATION REPORT
2  [tank1.coml;1, troop1.coml;2] − (1;2)(2;1) 1. Communication: [pass] 2. Constraint
3  process validation : [pass] 3. Meta−simulation validation : VP: [pass] Liveness
4  [tank1]: alive! Liveness [troop1]: alive! 4. Perfect model validation
5  Mathematical Composability: [pass] Exact match: Result: FALSE Runtime: 6291ms
6  Vepsilon (epsilon = 0.25): Exact match: FALSE Less than: TRUE Greater than: FALSE
7  Runtime: 16149ms
```

Figure 9. Validation Report

Discussion 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. To determine the runtime increase of the data-driven simulation model compared to a non-data driven simulation, we analyze the execution runtime of simpler queueing network models. In the Queueing Networks (QN) application domain, components are not data-driven, have at most two states and a reduced number of attributes. To determine the effect of the component complexity on the runtime, we compare with a single-server queue simulation model, because the number of components in this model is the closest to the number of components in the Tank vs Soldier Troop simulation model (the composition grammar of the QN application domain forbids models with less than three base components). Table 4 presents our findings. While an entirely fair comparison between the above models cannot be made, it is

	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
$\mathbf{V}_\varepsilon$	2.7	16.1

Table 4. Execution Runtime Evaluation

still clear that the validation runtime is increased for the data-driven models with large number of attributes and complex state machines. This complexity is inherent in the application domain and cannot be mitigated by further abstractions (of course, components in the Queueing Networks application domain could be as complex). For roughly the same number of components, the data-driven model has twice the number of attributes and incurs a runtime almost eight times higher. However, the runtime penalty is still acceptable considering the increased expressivity gains.

The Tank vs SoldierTroop example example raises some interesting issues. Firstly, we define a valid model as one that is *close enough* with respect to the states, sequence and duration of component execution, to a perfect model. Yet, what exactly is close enough (i.e. the values of ε), as with all thresholds, remains an open problem. Furthermore, the impact of different values of the unfolding grade τ remains to be studied. Intuitively, τ should be *large enough to capture all deviating behaviors*, but an optimal value of τ is difficult to obtain beforehand. Next is the difficult problem of defining the perfect models and components. While it is acceptable to assume their existence, how they are obtained is still an open question. For example, perfect components could be de-composed from perfect models specified by the simulation composer, they can exist a priori as we previously suggested, or the simulator composer can provide an ideal state machine for each individual component.

4. Related Work

Petty and Weisel pioneered a formal theory of composability validation which allows for a composed simulation model to be checked for semantic validity [19]. A composed simulation is modeled as the composition of mathematical functions that represent components over one-dimensional integer domains. The composed simulation is represented as a Labelled Transition System(LTS) where nodes are model states, edges are function executions, and labels are model inputs. However, time is not modelled and the function representing a component is assumed to be 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 representative of complex compositions. In contrast, we propose a new formal component definition where states change over *time*. Based on this definition, we represent the dynamic change of the entire simulation over time. To provide a more accurate measure of validity, we consider semantically related composition states in the definition of semantic relation V_ε.

DEVS (Discrete Event System Specification) [30] is a formalism derived from general system theory and is designed to describe the structure and behavior of a system. In DEVS, a system is modeled as a blackbox with state, input and output ports. For validation, compositions of DEVS models are represented in the Z specification language [27]. A theorem proving tool based on Z such as Z/EVES [20] is used to ver-

ify 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 more informal approach to composition validation uses the Base Object Model (BOM) as a component abstraction [18]. A BOM captures component behavior information including participating entities and their state machines, and information about the possible usage scenarios of the component. This approach assumes that a detailed user specified composition scenario exists to represent a valid composition. The scenario includes the sequence of component execution, as well as events and parameter names for interacting components. Component discovery is done 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.

Several approaches to validation exist in the software engineering community. In [4], component behavior is described using a finite automata-based notation. Individual component automata are composed to form a composite automaton. Subsequently, desired properties are verified using the alternation-free μ calculus. Unfortunately, the approach suffers from state explosion. Showcasing the expressivity of the Promela specification language, Java source code is transformed in Promela in the Bandera toolset [8]. However, extremely large state spaces form when complex software units are tested. In Wright [1], a component specification is composed from interface and computation parts. The interface consists of ports which define separate interactions in which the component will participate. Interconnections among component interfaces are made through connectors. Component computation and connector behavior are described in a CSP-like notation. Darwin [16] uses a component model with hierarchically nested components that are defined in terms of provided and required interfaces. Connections among components are plain bindings and no connectors are considered. Darwin allows dynamic reconfiguration via lazy and direct dynamic instantiation of components. In the lazy dynamic instantiation, a component with a provided interface is not instantiated until the first usage of such an interface. Direct dynamic instantiation allows defining architectures that dynamically evolve in an arbitrary way. Components and bindings are described using π calculus. However, most of the software engineering approaches described above focus on the structure on the structure of the composition or on the component coordination. This perspective does not fit well with modeling and simulation, where the focus is on behavior closeness to the real system.

5. Conclusions

Data-driven military simulations have complex behavior that changes rapidly with received data. To support component-based simulation and its semantic validation, a new data-driven component representation is proposed for specifying state transitions and attributes changes. Most military simulations are manually validated using a lengthy and costly multi-step process. We propose to address this using a two-step automated semantic composability validation approach. Firstly, we validate a component-based model for general properties including safety and liveness for instantaneous transitions and through time. In addition, safety and liveness are considered from a software verification perspective as logical statements, and from a simulation perspective in terms of output data and the desired composed model state changes. We show that a composed model consisting of components at higher level of abstraction can be easily translated into a concurrent process specification for verification using a model checker. However, real-life components may require lower-level of abstraction and higher expressivity to support validation in different contexts. To increase the validation accuracy, in the next step we validate the composed model with a perfect model. Components are represented using a new time-based formalism, and bisimulation is used to reason the composition validity. As expected, data-driven model validation incurs higher runtime but the overhead is small for the examples used.

While this paper addresses the semantic validation of data-driven simulation with *base components*, the complexity and scalability of component-based models can be increased by reusing the composed model as

a *model component* in new models. This involves extending our current formal validation process. However, addressing the important design trade-off between the degree of model component opacity and the accuracy of validation is an open question.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD Thesis, Carnegie Mellon University, 1997.

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

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

[4] T. Barros, L. Henrio, and E. Madelaine. Verification of Distributed Hierarchical Components. In *Proc. of FACS*, pages 41–55, 2005.

[5] R. Bartholet, D. Brogan, P. Reynolds, and J. Carnahan. In Search of the Philosopher's Stone: Simulation Composability Versus Component-Based Software Design. In *Proc. of the Fall Simulation Interoperability Workshop*, Orlando, USA, 2004.

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

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

[8] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, S. Laubach, and H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proc. of ICSE*, pages 439–448, Limerick, Ireland, 2000.

[9] P. Davis and R. Anderson. Improving the Composability of Department of Defense Models and Simulations, 2003.

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

[11] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. CAV'07*, pages 158–163, Berlin, Germany, 2007.

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

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

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

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

[16] J. Magee and J. Kramer. Dynamic Structure in Software Architecture. In *Proc. of SIGSOFT*, pages 3–14, San Francisco, USA, 1996.

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

[18] 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 *Proc. of DS-RT*, pages 145–155, Crete, Greece, 2007.

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

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

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

[22] C. Szabo and Y. Teo. On Syntactic Composability and Model Reuse. In *Proc. of the International Conference on Modeling and Simulation*, pages 230–237, Phuket, Thailand, 2007 (invited paper).

[23] C. Szabo and Y. Teo. An Approach for Validation of Semantic Composability in Simulation Models. In *Proc. of the 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 3–10, Lake Placid, USA, 2009.

[24] C. Szabo, Y. Teo, and S. See. A Time-based Formalism for the Validation of Semantic Composability. In *Proc. of the Winter Simulation Conference*, page (to appear), Austin, USA, 2009.

[25] Y. Teo and C. Szabo. CODES: An Integrated Approach to Composable Modeling and Simulation. In *Proc. of the 41st Annual Simulation Symposium*, pages 103–110, Ottawa, Canada, 2008.

[26] A. Tolk. What Comes After the Semantic Web - PADS Implications for the Dynamic Web. In *Proc. of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 55–62, Singapore, 2006.

[27] M. K. Traore. Analyzing Static and Temporal Properties of Simulation Models. In *Proc. of the Winter Simulation Conference*, pages 897–904, Monterey, USA, 2006.

[28] 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.

[29] US Department of Defense - VV&A Recommended Practices Guide. http://vva.msco.mil, (retrieved Aug. 2009).

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