Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
88(6) 707–730
© The Author(s) 2012
Reprints and permissions:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/0037549711428233
sim.sagepub.com
SAGE SCS

Supporting dynamic simulations with Simulation Modeling Architecture (SiMA): a Discrete Event System Specification-based modeling and simulation framework

Fatih Deniz[1], M Nedim Alpdemir[2], Ahmet Kara[2] and Halit Oğuztüzün[1]

Abstract

In this paper, we present our approach to introduce dynamism support to simulation environments, which adopts a Discrete Event System Specification (DEVS)-based modeling and simulation approach and builds upon previous work on Simulation Modeling Architecture (SiMA), a DEVS-based simulation framework developed at TUBITAK UEKAE. In the relevant literature there are already proposed solutions to the dynamism support problem. One particular contribution offered in this study over previous approaches is the systematic framework support for post-structural-change state synchronization among models with related couplings, in a way that benefits from the strongly typed execution environment SiMA provides. In addition to introducing theoretical extensions to basic SiMA, we report the results of performance measurements to illustrate the added value of dynamism extensions over the basic version, using a sample wireless sensor network simulation.

Keywords

Discrete Event System Specification, modeling and simulation, variable structure models

1. Introduction

Analyzing the behavior of complex and adaptive systems through simulation often requires the underlying modeling and simulation approach to support structural and behavioral changes. This requirement may stem from the inherent nature of the real world system under study, such as ecological or social systems,[1] it may stem from the modeling and simulation methodology of the analyst or it may be due to the way system modelers approach the modeling of inherent behavioral complexity of their models. A good example of a combination of the latter two is the case where the simulation study involves a large number of highly complex systems, the analyst wants to observe the behavior of these systems at varying levels of fidelity, and the modeler constructs the models in a way to allow the models to exhibit different observable behaviors during the course of simulation. This particular case implies that models may switch between different behavioral specifications (e.g. fidelity levels) dynamically at run time, depending on various triggering events. Some of the other possible reasons why dynamism support is required in a modeling and simulation framework can be listed as follows.

1. Some simulation scenarios can efficiently be executed only through dynamic structure support. For example, consider a simulation scenario in which two planes follow the terrain at a specific altitude. The provision of the terrain information to plane models could be implemented using an environment model. When this scenario is executed in a high-resolution setting, it may be impossible to load the terrain that represents the whole world at once. So, if the simulation requires the whole world, terrain has to be decomposed into regions, such that the environment model representing each of these regions interacts with the models representing the planes flying over it only when the plane enters that particular region.

[1]Middle East Technical University, Ankara, Turkey
[2]TÜBİTAK BİLGEM UEKAE/İLTAREN, Ankara, Turkey

Corresponding author:
Fatih Deniz, Sakarya Mahallesi, Cira Sokak, No:4/1, Altindag, Ankara, Turkey
Email: fatih.deniz@tcmb.gov.tr

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

2. Dynamism support can be used to yield optimal model execution. Adding dynamism support is most likely to increase performance. A model can be loaded into the memory whenever needed and can be removed from the memory after completing its job. Continuing from the previous plane example, in order to use the resources effectively, unnecessary sections of the terrain can be removed from the memory. In addition, through dynamic management of couplings and ports, unnecessary message transfers can also be eliminated, thereby increasing the performance by doing less work in each step.
3. Dynamism support may become a necessity for creating more realistic simulations. Model structure may have to be changed whenever necessary while the simulation is running. For example, if a missile explodes, the model representing it should also be removed from the model structure. Dynamism support is the natural way of handling this kind of situation.
4. Complex systems that require behavioral or structural changes to adapt to changing situations can be modeled more efficiently with variable structure models. Examples of these systems include wireless sensor networks (WSNs), distributed computing systems and ecological systems.
5. In addition, there may be unpredictable changes that need to be modeled at run time. For instance, consider a human population simulation, in which civilians and combatants are represented by different models. Under certain circumstances, a civilian can change into a combatant due to triggering events. The modeler may wish to design two distinct behavioral models for a combatant and a civilian, which requires dynamic switching from one model to another at run time. In order to allow simulations to adapt to these types of unpredictable changes, dynamism support can be used.

Allowing modifications to model structures and to internal functional specifications while the simulation is running, however, is a challenging task due to instabilities and inconsistencies this may introduce, especially if the underlying modeling approach does not provide a sound formal basis upon which the run-time infrastructures can be established. In this study, we take a particular stand to the problem of dynamism support in simulation environments by adopting a Discrete Event System Specification (DEVS)-based modeling and simulation approach and by building upon our previous work on Simulation Modeling Architecture (SiMA),[2,3] a DEVS-based modeling and simulation framework developed at TUBITAK UEKAE.

We observe that several approaches to dynamism are already proposed in the relevant literature.[1,4–7] We note that three distinct categories of change are discussed in those existing approaches: (a) a change in the overall compositional state of models; (b) a change in the connectivity relationships (coupling) among the models; (c) a change in the internal functional behavior of the model. We find two of the formal approaches to the variable structure models in the DEVS environment particularly relevant to our work. The first one is DSDEVS (Dynamic Structure DEVS), introduced by Barros.[5] The second one is Dynamic DEVS (DynDEVS), introduced by Uhrmacher.[1] A brief introduction to both approaches is provided in Section 3. In addition to these formal extensions, there are approaches that adopt existing formal specifications but contribute through different routes. For instance, Shang and Wainer[8] extend their existing simulation engine by adopting a combination of DSDEVS and DynDEVS. Similarly, Hu et al.[9] take a software engineering-oriented stand and propose a component-based simulation environment.

Our contribution to the work in this field is extending SiMA with dynamism support, building upon our basic SiMA-DEVS formalism. In particular, we formally present the extensions to SiMA-DEVS, called dynamic SiMA, to support structural dynamism, we present the extensions we have added to the implementation of SiMA-DEVS to achieve structural dynamism and we provide experiment results where dynamic SiMA is used. Our approach to add dynamism to our basic DEVS model is derived from that of DynDEVS. However, we do not have some of the operational limitations that DynDEVS has. Unlike DynDEVS, our approach allows dynamic port management and allows atomic models to initiate changes other than changing themselves only. One particular contribution we offer is the formal specification and systematic framework support for post-structural-change state synchronization among models with related couplings. This operation works in the opposite direction of the normal message flow and enables newly added models and newly added couplings to acquire the current state of the simulation. The details of our contributions are provided in Section 4.

The rest of the paper is organized as follows. Section 2 summarizes the relevant background work, Section 3 summarizes previous efforts for incorporating dynamism support into DEVS-based modeling and simulation environments, Section 4 provides a formal discussion of our approach and some details of the implementation, Section 5 describes a case study and presents our findings on the performance measurements and finally Section 6 provides conclusions and our plans for future work.

2. Background

2.1 DEVS formalism

The DEVS is a formalism introduced by Bernard Zeigler in 1976 to describe discrete event systems. In this formalism, there are two types of models: atomic models and

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

coupled models. Atomic models are basic units that specify behavioral logic. On the other hand, coupled models consist of other models (atomic and/or coupled) and connections between those models, but no behavioral specification. An atomic model in parallel DEVS formalism consists of a set of input events, a state set, a set of output events, an internal, external and confluent transition function and an output and time advance function.[10] The formal definition of an atomic model M with the set of input ports, *InPorts*, and the set of output ports, *OutPorts*, is as follows:

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

where

$X = \{(p, v) | p \in InPorts, v \in X_p\}$ is the set of input ports and values, where X_p is the set of values that can be received through port p,

$Y = \{(p, v) | p \in OutPorts, v \in Y_p\}$ is the set of output ports and values, where Y_p is the set of values that can be sent through port p,

S is the set of states,

$\delta_{int} : S \to S$ is the internal state transition function,

$\delta_{ext} : Q \times X \to S$ is the external state transition function such that:

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

e is the elapsed time since the last transition,

$\delta_{con} : S \times X \to S$ is the confluent transition function,

$\lambda : S \to Y$ is the output function,

$ta : S \to R^{+}_{0, \infty}$ is the time advance function.

In DEVS formalism, model interaction semantics are encapsulated in the notion of ports that define externally visible interfaces of models. Input ports receive external input events X, and output ports send output events Y. An atomic model has a state set S, where each state $s \in S$ is associated with a time value, calculated by a time advance (ta) function, which determines the maximum duration of the state. If no external event is fired during this time, the first output function (λ) and then the internal transition function (δ_{int}) is executed. If the model receives an external event during this time, then the external transition function (δ_{ext}) is executed. The current state of the model is updated in internal and external transition functions. If the model receives events at the time of its internal transition, then the confluent transition function is executed. This function is the main difference between parallel DEVS and the classical DEVS definition.

Coupled models do not contain any behavioral logic. They define the overall model composition structure, specifying both port couplings and the hierarchy between constituent models. A coupled model in parallel DEVS formalism is defined formally as follows:

$$CM = \langle X, Y, D, \{M_i\}, EIC, EOC, IC \rangle$$

where

$X = \{(p, v) | p \in InPorts, v \in X_p\}$ is the set of input ports and values, where X_p is the set of values that can be received through port p,

$Y = \{(p, v) | p \in OutPorts, v \in Y_p\}$ is the set of output ports and values, where Y_p is the set of values that can be sent through port p,

D is the set of component names,

M_i is the model of component i, for $i \in D$

EIC, EOC and *IC* define the coupling structure,

EIC is the set of couplings between input ports of the coupled model itself and input ports of its components,

EOC is the set of external output couplings, which connect its components' output ports to the model's own output ports,

IC is the set of internal couplings, which connect a component's output port to another component's input within the coupled model.

A complete description of DEVS semantics can be found in Zeigler and Praehofer,[10] Zeigler[11] and Barros et al.[12]

2.2 SiMA formalism

SiMA[2,3] is a modeling and simulation framework that is based on the DEVS approach as a solid formal basis for complex model construction. SiMA Simulation Execution Engine implements the parallel DEVS protocol, which provides a well-defined and robust mechanism for model execution. SiMA builds upon a specialized and extended form of DEVS formalism, namely SiMA-DEVS, which does the following.

1. Formalizes the notion of 'port types', leading to a strongly typed (and therefore type-safe) model composition environment. In this respect, we specialize the basic DEVS formalism by introducing type-system driven syntactic constraints on the port definitions.
2. Introduces a new function as a first class construct to allow for state inquiries between models with possible algebraic transformations (but no state change), without simulation time advance. In this respect, we extend the definition of an atomic model in the DEVS formalism.

Given an Extensible Markup Language (XML) Schema-type system Γ, an atomic model M with sets of input and output ports, *InPorts*, *OutPorts*, respectively, SiMA-DEVS formalism is defined as follows:

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

where

S, δ_{int}, δ_{ext}, δ_{con}, λ, ta are as defined in the parallel DEVS formalism given in Section 2.1, $X = \{(p, v) | p \in InPorts, v \in X_p\}$ is the set of input ports

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

and values, where X_p is the set of values that can be received through port p,

$Y = \{(p, v) | p \in OutPorts, v \in Y_p\}$ is the set of output ports and values, where Y_p is the set of values that can be sent through port p,

Inports, Outports are the sets of input and output ports such that:

$$InPorts = \{(\Gamma, \tau) | \Gamma \vdash v : \tau, v, \in X_p\}$$

$$OutPorts = \{(\Gamma, \rho) | \Gamma \vdash v : \rho, v, \in Y_p\}$$

Γ : is the XML Schema-type system,
τ, ρ : are data types valid with respect to the XML Schema-type system, $\delta_{df} : PDFT_{in} \times S \rightarrow OutPorts'$ is the time invariant direct feed-through function, where $PDFT_{in} \subseteq InPorts$ and $OutPorts' \subseteq OutPorts$.
Note that the set of input ports, *Inports*, is formally defined as a set of pairs where each pair defines one input port of a model uniquely. The first element of each pair, Γ, is a typing environment (in our case the XML Schema-type system) and the second element of the pair (τ) is a data type that is valid in Γ, where each input value v conforms to data type τ. This is formally denoted by the typing judgment $\Gamma \vdash v : \tau$, which asserts that a term v has a type τ with respect to a static typing environment Γ for the free variables of v (or shortly ' v has type τ in Γ ').[13] Similar semantics apply to output ports, too. Thus, we make *strong typing* and *type-system dependency* of the ports explicit in the formal model. It may be argued that strong typing and type-system dependency are essentially run-time properties of the execution environment, and that incorporation of these aspects into the definition may ambiguate the abstraction level of the formal specification. Although this argument may be valid in many cases, we counter-argue that there are a number of merits our proposed route, in particular the following.

1. We introduce a type discipline to the definition of the externally visible model interfaces (i.e. ports) leading to a sound basis for the specification of the overall information model of the system being modeled. Ensuring that the overall information model is specified using a well-defined type system we support syntactic compatibility at the modeling level.
2. We facilitate model-driven engineering through well-typed and type-system dependent external plugs to enable automated port matching and model composition. In fact, a model-driven simulation construction tool chain for SiMA is successfully implemented, via a number of tools, such as a code generator, a model builder and a model linker. Thus, by making strong typing and type-system dependency explicitly visible in the formalism, we reduce the gap between modeling-level logical composability constraints and run-time level pluggability constraints.
3. Model implementation can be supported via a type-safe input/output (I/O) data handling and state management mechanism that eases the model development process and enables compile time consistency checks. In fact, our SiMA implementation provides a template class for atomic model developers that maintains typed object and event managers. An object manager is a template vector initialized with the data type of an input or an output port it is bound to. Thus, each port is bound to an object or event manager and a model developer can readily access and manipulate arbitrarily complex data flowing via the ports in a type-safe manner.

It is worth noting that, in its current form, the notion of strongly typed ports comes with an obvious limitation: port couplings need to be based on strict type equality. In other words, model composability is constrained with syntactic type matching. Presumably, this limitation can be alleviated by introducing semantic mapping mechanisms to allow for type equivalence (rather than strict equality), but currently this is not supported.

Note also that, in addition, SiMA-DEVS introduces a new function, δ_{df}, that enables models to access the state of other models through a specific type of port, without advancing the simulation time. As such, it is possible to establish a path of connected models along which the models can share parts of their state and use state variables to compute derived values instantly within the same simulation time step. This is similar to the notion of zero-lookahead in high-level architecture (HLA).[14] One may argue that the zero-lookahead behavior could be modeled by adjusting the time advance function of an atomic model such that the model causes the simulation to stop for a while, do any state inquiry via existing couplings, then re-adjusting the time advance to go back to the normal simulation cycle. Although this is possible, we hold that by introducing a function and a specific port type that is tied (through run-time constraints imposed by the framework) to that particular function several advantages are gained:

1. the models can communicate and exchange part of their state with each other without the intervention of the simulation engine, thus providing a very efficient run-time infrastructure;
2. allowing such communications only to occur through a specific port type, the framework is able to apply application-independent loop-breaking logic at the ports to prevent algebraic loops, thereby ensuring model legitimacy.

For example, in Figure 1 the platform model is just updating its X and Y coordinates and does not deal with the

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

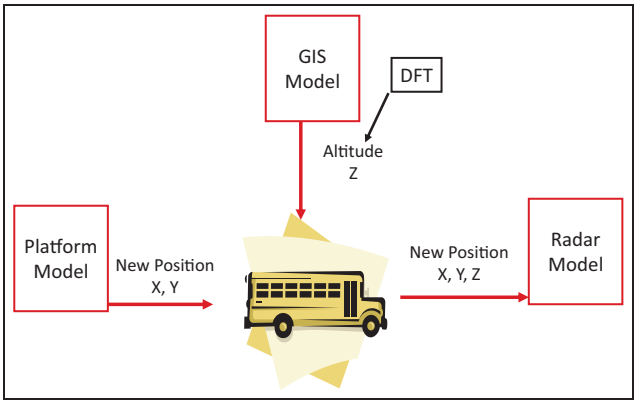

Figure 1. SiMA δ_{df} example use.

object's altitude. However, the Radar Model requires the platform object's *Z* coordinate in order to generate the correct radar behavior. By using the δ_{df} function, the Geographic Information System (GIS) model can calculate the altitude of the platform object and send it to the Radar Model within the same time step of the platform update, so that the Radar Model obtains all of the *X, Y* and *Z* properties at the same time and calculates its estimations accordingly.

Intuitively, we can state that the behavior of SiMA-DEVS is equivalent to that of parallel DEVS, from a language equivalence point of view.[15] Hwang[15] shows that two atomic DEVSs can be equivalent if the languages they generate are the same. The total language a DEVS atomic model generates is simply defined in terms of transition trajectory, output trajectory and the total trajectory generated by that DEVS model. Since the new function we introduce does not change the model's state, the state trajectory would be the same as parallel DEVS. Although the output trajectory of a SiMA-DEVS model is directly affected by δ_{df}, the same output behavior can be generated by a parallel DEVS model via the next time calculation logic. In other words, the parallel DEVS models can be forced to stop advancing at a specific point in time to send data to other models. As such, equivalence is also true from a timed language[16] point of view. So, the benefit SiMA-DEVS offers in this respect is that the ability to model zero-lookahead behavior is encapsulated into a well-defined mechanism and this mechanism is introduced into the model definition as a first class entity, rather than relying on custom implementations. We leave the formal discussion of the language equivalence out of the scope of this paper and refer the reader to Hwang,[15,16] and Hwang and Cho.[17]

2.3 SiMA software architecture

In a nut shell, SiMA provides a software framework for developing simulation models and an engine for executing simulations. As can be observed in Figure 2, SiMA consists of two main layers: *SiMA Core* and *C++ Interface*. Arrows in the figure denote data and interaction dependencies between layers and components.

The *SiMA Core* consists of five sub-modules that are used for modeling and simulation. The *Modeling Framework* is a set of classes and data types to be used in model development. Atomic models and all their

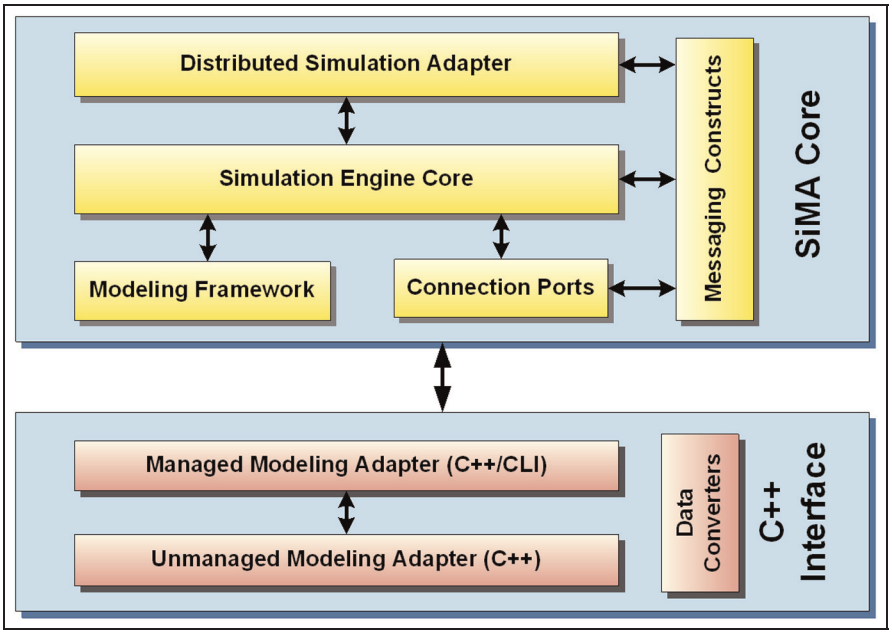

Figure 2. SiMA software architecture.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

subclasses are defined in this framework. *Connection Ports* is the transportation component that contains classes for defining ports and their connection characteristics. The *Simulation Engine Core* includes the functionality for executing simulations and in addition exposes administrative interfaces to manage and track the simulations at run time. It contains a simulator component that encapsulates most of the functionality required to implement the parallel DEVS simulation protocol. The Simulation Engine Core module uses the *Modeling Framework* and *Connection Ports* to fulfill its functions. The *Distributed Simulation Adapter* is the interface for connecting SiMA simulations to external distributed simulation infrastructures, such as HLA compliant Run-time Interfaces (RTIs). *Messaging Constructs* is the component that presents all base data type classes and rules for inter-communication of atomic models and *SiMA Core* components.

The *C++ Interface* layer allows C++ to be used as a model implementation language for SiMA atomic models. All core SiMA components are developed in .NET, but SiMA supports models implemented in both .NET (C#, Managed C++) and pure C++ to co-exist in the same run-time environment during a simulation. However, since there is a strict boundary between their coding environments, various adapters and components that manage the interoperability between .NET and C++ atomic models and SiMA components are implemented in the *C++ Interface* layer.

The *C++ Interface* layer consists of three sub-components. The *Unmanaged Modeling Adapter* has the same interface and class hierarchy as the .NET Modeling Framework, except it is developed in pure C++ language. The *Managed Modeling Adapter* is developed in C++/CLI, which is a special edition of C++ language in .NET that allows access to both C++ and .NET methods and data types. The *Managed Modeling Adapter* handles the interoperability management and delegates all simulation commands to C++ models, and provides all information required by them from the simulation environment. *Data Converters* are special adapters that perform marshalling of all values between .NET and C++ data types in both ways. A model developer can use the KODO tool to auto-generate these data converters for his/her data types.

3. Related work

In the relevant literature, there are two main approaches to the dynamism support problem, namely DSDEVS and DynDEVS. In this section a brief introduction to these formalisms will be given.

3.1 DSDEVS

DSDEVS was introduced by Barros in 1995.[5,18,19] In this approach, the atomic model definition remains the same as classic DEVS formalism, but the classical coupled model definition is extended. Each coupled model is associated with a dynamic structure atomic model that handles the structural changes in its associated coupled model.

A DSDEVS network model is formally described as follows: $DSDEVSN = \langle X_\Delta, Y_\Delta, \chi, M_\chi \rangle$, where Δ is network name; χ is the name of the DSDE network executive; M_χ is the model of χ; X_Δ is the set of input events; and Y_Δ is the set of output events. M_χ, the model of the network executive χ, is a basic DSDE model defined as $M_\chi = \langle X_\chi, Y_\chi, S_\chi, \delta_{int_\chi}, \delta_{ext_\chi}, \lambda_\chi, ta_\chi \rangle$

M_χ contains information about network composition and coupling. A state $s_\chi \in S_\chi$ has information about the structure of the network model and it is defined as $s_\chi = \left(D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT, V\right)$, where D is the set of component names; M_i is the model of component i, for $i \in D$; I_i is the set of component influencers of i, $\forall i \in D \cup \{\chi, \Delta\}$; $Z_{i,j}$ is the i-to-j output to input translation function, for all elements of I_i; *SELECT* is the tie-breaker function; and V represents other state variables of the network executive.

In DSDEVS, only the network executive can make structural changes and any change made in one of the components of the 5-tuple $\left(D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, SELECT\right)$ will be automatically reflected to the structure of the network model. The DSDE formalism developed subsequently by Barros et al. is a modified version DSDEVS and it defines its behavior in a parallel way, making networks of models more amenable to an implementation in a parallel machine. A detailed explanation of DSDE formalism can be found in Barros,[19] and abstract simulators necessary to simulate DSDE models can be found in Barros.[20]

3.2 DynDEVS

The DynDEVS approach was introduced by Uhrmacher in 2001.[1] Unlike DSDEVS, DynDEVS formalism does not introduce a dedicated type of model (i.e. the network executive model) to apply structural changes dynamically. Instead, transition functions ρ_α and ρ_N are added to the atomic and coupled model definitions, respectively, to support dynamism. There are two types of models defined in DynDEVS formalism, which are dynDEVS and dynNDEVS models. Atomic models in DynDEVS formalism are formally defined as follows: $dynDEVS = \langle X, Y, m_{init}, M(m_{init}) \rangle$, where X, Y are structured sets of inputs and outputs; $m_{init} \in M(m_{init})$ is the initial model; $M(m_{init})$ is the least set having the structure $\langle S, s_{init}, \delta_{int}, \delta_{ext}, \lambda, ta, \rho_\alpha \rangle$, where S is the set of states; $s_{init} \in S$ is the initial state; δ_{int}, δ_{ext}, λ, ta are the same functions as in classical DEVS formalism; and $\rho_\alpha : S \to M(m_{init})$ is the model transition function.

Coupled models are described in DynDEVS formalism formally as follows: $dynNDEVS = \langle X, Y, n_{init}, N(n_{init}) \rangle$, where X, Y are structured sets of inputs and outputs;

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

$n_{init} \in N(n_{init})$ is the start configuration; $N(n_{init})$ is the least set having the structure $\langle D, \rho_N, \{dynDEVS_i\}, \{I_i\}, \{Z_{i,j}\}, Select\rangle$, where D is the set of component names; $\rho_N : S \to N(n_{init})$ is the network transition function with $S = \times_{d \in D \oplus m \in dynDEVS_d} S^m$; $dynDEVS_i$ is the DynDEVS model with $i \in D$; I_i is the set of influencers of i; $Z_{i,j}$ is the i-to-j output–input translation function; and *Select* is the tie-breaking function.

In DynDEVS formalism, there are certain operational constraints. Just like atomic models, coupled models, too, cannot make structural changes outside their enclosing model. Another constraint is that dynamic port management in the sense of addition and removal of ports to existing models is not supported. Allowed operations include insertion and deletion of models, and addition and removal of couplings between existing ports. Further details on DynDEVS formalism can be found in Uhrmacher[1] and Himmelspach and Uhrmacher.[21]

4. Our approach

4.1 Adding dynamism to SiMA-DEVS

Our approach to add dynamism to our basic DEVS model is based on to that of DynDEVS, as indicated earlier. To be more precise, we conform to both dynDEVS and dynNDEVS definitions as the underlying formal specification, with some extensions that are elucidated below.

1. Structured sets of inputs and outputs are defined in conformance to our strongly typed port definitions, where the formal definitions for P_{in}, P_{out} apply; $M(m_{init})$ is the least set having the structure $\langle S, s_{init}, \delta_{int}, \delta_{ext}, \lambda, ta, \rho_\alpha, \delta_{df}\rangle$, where $S, s_{init}, \delta_{int}, \delta_{ext}, \lambda, ta, \rho_\alpha$ are the same as in DynDEVS formalism; δ_{df} is the direct feed-through transition function defined in SiMA-DEVS. Thus, the meaning of a *model* in SiMA-DEVS is aligned with that of DynDEVS and the top-level semantics of the DynDEVS definition are maintained. In other words, our dynamism mechanisms are non-disruptive to the overall semantics of the basic DynDEVS formalism.
2. We introduce a state synchronization mechanism between connected models, to be performed at the end of a structural change phase, in case a model is required to update the values of those state variables that are within the common set of pre- and post-change models (i.e. they are not introduced newly after the model's structural transition) but have values that remained unchanged during the pre-change simulation period. This mechanism can be instrumental in several specific cases. Consider, for instance, two models A and B where model A is the *influencee* (affected by the output of model B) and model B is the *influencer* (its output ports are connected to the input ports of model A). Suppose model A initializes some of its state variables at the beginning of simulation but does not receive updates for these variables until model B goes through some structural change. In this case model A can use our state synchronization mechanism to update the relevant portion of its state. Another case arises when model B is added in the middle of a simulation and introduces new couplings influencing the input ports of model A. Once again, model A needs to synchronize with model B to update its state. One might argue that, after the structural change, synchronization of such state variables would already take place as a result of message passing via the coupling links during the normal course of the simulation. However, it is important to note that, due to differences in state update rates (i.e. different step sizes), an influencee may have to go through many state updates and produce many output sets before it can receive the required updates from slower influencers, a case that might potentially lead to significant errors in the behavior of the overall simulation, especially if the simulation application requires a high degree of behavioral sensitivity. It is worth mentioning that the state synchronization function must be executed as the last step of the structural change transition phase to allow the influencers to perform the necessary state updates before the influencees ask for the latest values of the state variables that need to be synchronized.

A variable structure atomic model is defined formally as follows:

$$VS_{AM} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta, \delta_{df}, SO, \rho_\alpha\rangle$$

where $X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta, \delta_{df}$ are as in the basic SiMA-DEVS formalism, SO is the set of structure change operations and $\rho_\alpha : S \times SO \to S\prime$ is the structure change transition function. ρ_α defines a mapping from pre-change state and structure change operations ($S \times SO$) to post-change state ($S\prime$).

A variable structure coupled model is defined formally as follows:

$$VS_{CM} = \langle X, Y, n_{init}, N(n_{init})\rangle$$

where,

$n_{init} \in N(n_{init})$ is the start configuration,

$N(n_{init})$ is the least set having the structure

$$\langle D, M_i, I_i, Z_{i,j}, \gamma, \tau\rangle$$

where

D is the set of component names,

M_i is the model of component i, for $i \in D$,

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

I_i is the set of component influencers of i,

$Z_{i,j}$ is the i-to-j output-to-input translation function, for $j \in I_i$,

$\gamma : SO \times S^{N(n_{init})} \rightarrow S^{N(n_{init})}$ is the network change structure transition function, where $S^{N(n_{init})} = \times_{d \in D \oplus_{k \in M_d}} S^k$

in which $S^{N(n_{init})}$ is defined as the cross-product over the state space of its components. Since the internal state of a component might vary within the boundary of M_i, we define the state space of each component as a disjoint sum of the state spaces of its incarnations m_k. (A disjoint sum is a construction that takes two sets and combines them to obtain a set in which the two originals are embedded with no overlap and no superfluous elements. Equivalently, it is a set via which any pair of functions, one from each of the given sets, can be factorized. For a more detailed discussion of why a disjoint sum is used in this context see Uhrmacher.[1])

$\tau_i : S_{M_i} \rightarrow S'_{M_i}$ is the state synchronization function, where $S'_{M_i} = {}_{j \in K_i}\left\{ \prod_{L\sigma C} (S_j) \right\}$, where $K_i \subset I_i$.

In other words, τ_i computes the synchronized state (S'_{M_i}) for M_i, by applying selection (σ) and projection ($\mathbf{\Pi}$) operations on the states of some of the influencers of model M_i, and then producing the union of these states. In this definition, L denotes the reduced (projected) tuple for S_j and C denotes the condition of the selection.

In this definition, when structural change is initiated by top-level simulator, the γ transition function is executed. A structure change request initiated by the top-level simulator is disseminated to the children-coupled models recursively. Each coupled model, receiving the request, applies the structural changes relevant to it and passes the requests to its relevant children (i.e. those addressed by the request). On the other hand the network transition function is executed when structural change is initiated by a leaf-level atomic model. In this case, each coupled model collects request messages from its children, applies changes relevant to it and passes upper-level requests to its parent-coupled model. Further details of how a particular structural change is achieved through the mechanisms SiMA provides are presented in Section 4.3.

We now set out to describe the principles that govern the run-time algorithms of the SiMA simulation engine to manage structural changes. In SiMA-DEVS, there is no dedicated *executive* model that supervises atomic models as in the DSDE formalism. Although notionally SiMA coordinators play a similar role to the network executive model in that they accumulate change requests, they do not include model logic (i.e. transition functions, state, inputs and outputs, etc.) as in the DSDE network executive. In SiMA-DEVS, similar to DynDEVS, atomic models are responsible for initiating structural changes. This model-centric approach seems to be more reasonable, since most of the potential change-triggering events that require structural changes from a particular model are naturally handled by the external transition function of that atomic model. In addition, it is that particular model which should have the knowledge of re-structuring itself, whether this re-structuring is a switch to an internally defined different functional model, or a re-adjustment of its port couplings.

The only exception where the model-centric approach may become restrictive is the case where a new model (atomic or coupled) is to be added to the simulation. The logic for initiating a model insertion may require that more than one model contribute to the decision, or model insertion may be a user-initiated request that is not necessarily handled by a single model. In current implementations of DynDEVS, namely, AgedDEVS and JAMES, it is reported that the atomic models are assumed to have access to a knowledge base from where they can collect the necessary information to decide new model additions.[1] Although this approach seems quite reasonable for agent-oriented implementations, it introduces a dependency to a specific architectural scheme and behavioral semantics for simulation applications, which we are inclined to avoid. Therefore, in our approach the following apply.

- If an atomic model requires a structural change, it informs its parent coordinator about the type and content of the operations to apply. Coordinators store all structure change requests until all child models complete their operations. These requests will be processed by the relevant coordinator. In the case of overlapping requests (i.e. requests involving change operations targeting the same model), those requests will be combined and propagated to the relevant model by the parent coordinator. An atomic model can create and send structure change requests to its parent-coupled model, but cannot change its own structure. A coupled model processes these messages and executes the operations restricted to its bounding coupled model and sends the requests targeted at outside its boundaries to its parent-coupled model. The target of the change is resolved by checking the path attribute found in the change request message (see Section 4.5 for more details on the content of a change request message).
- The software application that is hosting the simulation may initiate structural changes, too. This request is sent to the root coordinator to be executed over the model structure recursively. The root coordinator implements an interface that allows applications to send their structural change requests to the simulation engine. This operation is applied in two parts:
  - before applying the change operation, simulation is suspended at the beginning of the next cycle;

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

- the change request is processed by the root coordinator and child model operations are sent to the child coordinators recursively, causing all related child coordinators to apply change operations specified in the request.

4.2 Operations on model structures

There are three main types of structural change operations defined in SiMA: adding/removing a model, adding/removing a coupling and adding/removing a port.

- *Removing a model*: this operation consists of two steps: removing all the connections from/to the model, and removing the model.
- *Adding a model*: this operation consists of three steps:
  1. adding a model to the parent-coupled model;
  2. calling the init() function of the newly added model to initialize its state variables;
  3. calling the AdvanceTime (CurrentTime) function for synchronization.
- *Adding/removing a coupling*: adding a coupling is a critical operation in our case. After adding a coupling, a process for synchronizing the current states of newly connected models is executed. To achieve this, a data request mechanism between connected ports is implemented that operates in the opposite direction of the normal message flow. An input port creates a request and sends it to the newly connected ports. An answer to this request is generated and sent to the requesting port. These response messages will be handled when the external transition function of the model is executed.
- *Adding/removing a port*: this operation supports the addition of new ports to coupled models. Note that the new ports must conform to one of the existing port types. Before removing a port, all couplings from/to the port are removed.

Our framework does not support the addition and removal of new *port types* to the type space of the simulation at run time. One rationale for this is to preserve the models' external identity, as advocated by Uhrmacher.[1] Another important reason for such a restriction is the implied ambiguities in the run-time behavior of source and sink models of the newly added ports with new port types. To be more specific, say, for instance, a new output port of a new type is to be added. This would normally cause new connections to be established between its source and some other sink model. To be able to process the data coming from the new port type, the sink models have to be structurally and behaviorally ready to receive, interpret and process data coming from the new port. In a type-safe environment where port connectivity is regulated and restricted by type compatibility between connected ports (which is the case in SiMA), normally a new port will have to be added to the sink model too. However, both the source and the sink model may not know in advance the processing logic of the information flowing through these new ports. Such a support would rely on the pre-existence of sophisticated application-specific semantics within the models. We believe this case should be avoided for generic frameworks and, therefore, we exclude this functionality. However, unlike DynDEVS, we do find the addition of ports having a port type already defined in the current type space useful, since it allows one to establish a new coupling with an existing model.

For an example where some of these operations are applicable, consider a simulation scenario involving two planes flying in formation. A graphical representation of the models involved in this scenario can be seen in Figure 3. When the simulation starts execution, the models representing the planes send their states to each other from their ports once and subsequently they only send their current locations and directions, which are the only updated parameters of the planes throughout the simulation. Assume that at some point in time, a third plane is to be added to the simulation to connect to the existing planes. This updated model can be seen in Figure 4. Since the first two planes send only their updated parameters, which are

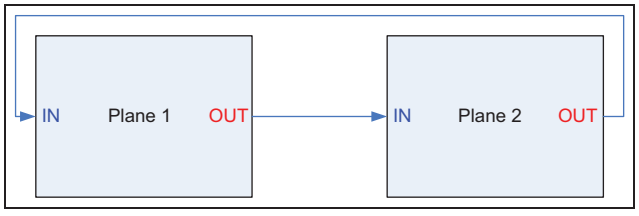

Figure 3. Initial model.

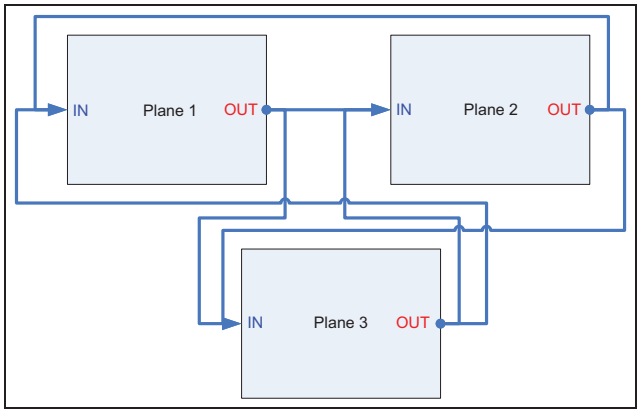

Figure 4. Updated model.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

location and direction, a newly added plane will not be aware of the remaining two planes' relevant part of their states. Therefore a state synchronization is required.

Dynamic SiMA handles this case by implementing an automated state synchronization mechanism via a querying system between connected port pairs. When a coupling is added to the model structure while the simulation is running, this querying system automatically works as a service provided by the infrastructure, without incurring any additional implementation overhead on the model developer. A more detailed discussion of the state synchronization mechanism is provided in Section 4.4.

4.3 SiMA abstract simulators adapted for dynamism support

Recall that SiMA is an implementation of SiMA-DEVS formalism as discussed at the beginning of this section. The SiMA run-time layer is implemented in C# programming language but it can interface to models implemented in both C++ and C# programming languages. To incorporate dynamism support into the SiMA run time, a number of modifications are applied. A summary of these modifications for supporting variable structure models is given below.

i. A property, named StructureChangeRequired, is added to the atomic models' simulators.
ii. Atomic models that may require structural changes while the simulation is running implement ρ_α transition function.
iii. The GetNextTime functions of the coordinators and simulators are modified so that, in addition to the minimum advance time value, they now return a StructureChangeRequired flag, too. To initiate a structural change, an atomic model simply sets its StructureChangeRequired flag to true.
iv. When a structure change request arrives at the root coordinator with the minimum advance time value, a structure change step is executed. For each atomic model that requires structural changes at the new current time, the ChangeStructureTransition function is executed.

In the remainder of this section, extensions to abstract simulators required for executing variable structure SiMA models are provided as code fragments described in pseudo code format. The code fragments include only critical parts of the processing logic. We first present an activity diagram in Figure 5 that denotes the high-level logic driving the execution of the code fragments.

4.3.1 Root coordinator. The algorithm that is executed at the top-level root coordinator is shown in Algorithm 1. It can be observed in this algorithm that there are two distinct cases in regard to the initiation of a structure change and three alternative simulation cycles can potentially be executed depending on these cases. If the structure change is initiated by an external request, the statements between lines 3–6 are executed, if the structure change is initiated by a leaf-level atomic model, statements between lines 13–15 are executed, otherwise the normal simulation cycle is executed.

```
1: while simulation-end-condition not satisfied do
2:     if structure change requested from top level then
3:         Process change request
4:         Send subrequests to related child coordinators
5:         Do state synchronization
6:         Initialize newly added models
7:     end if
8:
9:     <currentTime, changeReq> ←model.GetNextTime()
10:    Advance simulation time to currentTime
11:
12:    if changeReq is True then
13:        Execute a structural change step
14:        Do state synchronization
15:        Initialize newly added models
16:    else
17:        Execute a normal simulation cycle
18:    end if
19: end while
```

Algorithm 1. Root coordinator.

After a structural change operation, all models that have new couplings will execute the state synchronization mechanism discussed in Section 4.4 to update their state information.

The ChangeStructure function has a similar effect to that of the structure change (sc) message found in both DSDEVS[8,20] and DynDEVS.[21]

4.3.2 Coordinator. In the GetNextTime function of a coordinator, shown in Algorithm 2 the next simulation time and whether any structural change is required at that time is resolved recursively down the model hierarchy and the result is sent back to the parent coordinators up the hierarchy.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

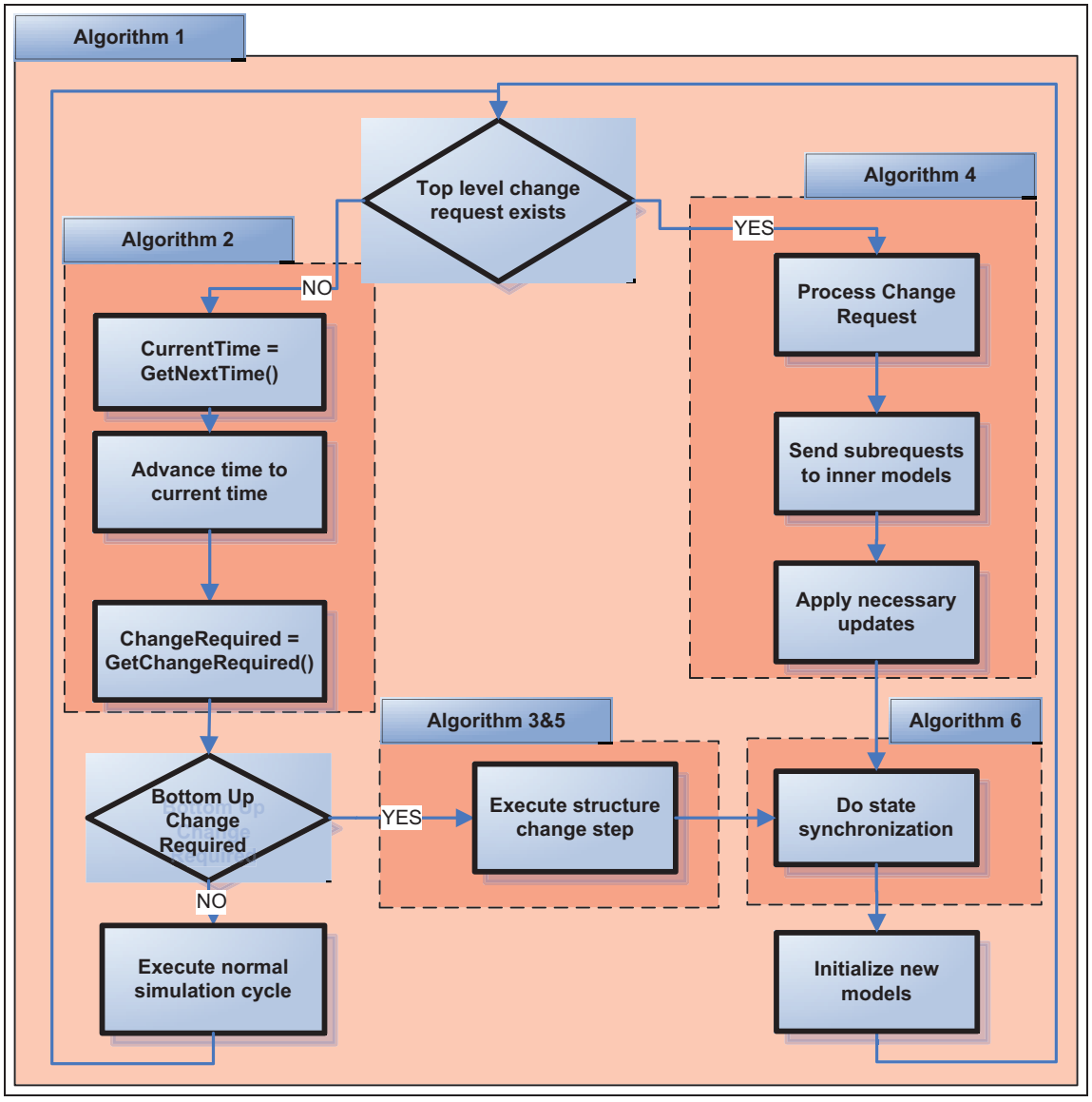

Figure 5. Top-level activity diagram.

1: **for each** inner model M **do**
2: <time, changeReq> ← M.GetNextTime()
3: **if** time < minTime **then**
4: minTime ← time
5: commonChangeReq ← changeReq
6: **else if** time = minTime and changeReq is True **then**
7: commonChangeReq ← True
8: **end if**
9: **end for**

Algorithm 2. Recursive next time calculation.

1: **for each** inner model *M***do**
2: **if** M requested structure change **then**
3: Call M's change structure function
4: Add M's change requests to changeReq set
5: **end if**
6: **end for**
7: Process changeReq set
8: Apply necessary updates in current level
9: Send upper-level operations to parent model

Algorithm 3. Structure change in coordinator -from the bottom.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

If a change-structure step is initiated by a leaf-level atomic model, in a coupled model's coordinator, Algorithm 3 is executed, which causes the initiating model to apply any model-specific logic, applies any required updates to other models at the same level as the initiator, and sends the change request up the hierarchy. On the other hand, if the step is initiated by the top-level root coordinator, Algorithm 4 is executed, similarly to the former but in the reverse direction of change request propagation. The basic idea in both algorithms is to apply necessary updates in the coupled model they are associated with and redirect the remaining requests to the models at lower or higher levels where they are addressed.

```
1: Process requests
2: for each inner coupled model C do
3:     if a request exists for model C or its submodels then
4:         Send related requests to model C
5:     end if
6: end for
7: Apply necessary updates in current level
```

Algorithm 4. Structure change in coordinator -from the top.

4.3.3 Simulator. The GetNextTime function of the simulator returns the next internal transition time and an indication of whether any structural change request exists at that time. As indicated in Algorithm 5, the structural change function of an atomic model (i.e. ρ_α) is executed if and only if its next time is imminent and a structural change request has been issued by that model.

```
1: if changeRequired is True and simulation time = t_N then
2:     Call ρ_α
3:     changeRequired ← False
4:     Send required change packages to parent model
5: end if
```

Algorithm 5. Structure change in the simulator.

If the state of an atomic model satisfies certain conditions that require structural changes, the atomic model marks itself as pending for change and informs its simulator to initiate a structural change process. This simulator then recursively sends the request to the root coordinator. Structural change requests can be issued by any atomic model during one of its transition functions. These requests are handled in the next internal transition phase.

4.4 State synchronization mechanism

One of the major extensions to realize dynamism support was to add a state synchronization mechanism between connected ports. With this extension, a port can create a request and send this request to other source ports to which it is connected. This mechanism works in the opposite direction of the normal message flow and enables newly added models or newly added couplings to acquire the current state of the simulation. This capability is crucial for SiMA, since ports are managed by event and object managers where object managers send only modified data for efficiency reasons. Therefore a sink model would not have up-to-date values of certain state variables from the source models if before the structural change the sink model did not use those particular state variables. If a model requires the previously updated fields, it can prepare and send a request to gather this information. Synchronization might also be required for cases where a model (call it an event initiator) sends an event to trigger a specific behavior of other models connected to its output ports. In such cases, if a newly added model is required to connect to this event initiator, it would require the last event (e.g. a triggering command) in order to synchronize the part of its state that would normally be updated via the external transition function when that particular event was received. Implementation details of state synchronization mechanism are discussed below.

An interface named IPortValueSource is defined in SiMA as shown in Algorithm 6.

```
1: RequestState(destModel:string, destPort:Port):Message[]
```

Algorithm 6. IPortValueSource interface.

This interface has only one member function, which takes a destination model and a destination port as input parameters and returns the port's related data. In a structural change step, after all structural updates are completed, the states of the models are updated accordingly, as shown in Algorithm 7. Each coupled model contains a list of couplings, in its level, that are added in the last structural change step.

```
1: for each inner coupled model M do
2:     Call state synchronization procedure of M
3: end for
4: for each new coupling c do
5:     Add destination port to the affected ports list
6:     Message[] messages ← c.SourcePort.RequestState(destination model, destination
port)
7:     for each Message m in messages do
8:         Put values of m into destination port
9:     end for
10: end for
11: for each affected port P do
12:     Synchronize sources of P
13: end for
14: Clear new couplings list
```

Algorithm 7. Updating states.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

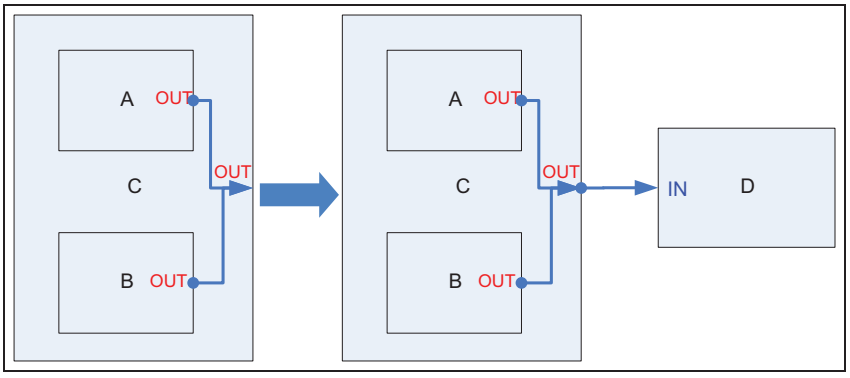

Figure 6. Dynamically adding a model and a coupling.

Destination ports create requests and send these requests to the source ports that are connected to them. When an atomic model receives a request, it sends part of its state that qualifies for the posed request as a response. When a coupled model receives a request, it redirects this request to the source ports that are connected to this port. Then, the coupled model collects and returns the responses received from those redirected ports. For example, in Figure 6 a model named *D* and a coupling from *C*'s *OUT1* port to *D*'s *IN1* port is added dynamically. In this example, the state synchronization process works as follows:

1) the IN1 port of model D sends a request to the OUT1 port of model C;
2) the OUT1 port of model C redirects this request to the OUT1 ports of model A and model B;
3) the OUT1 port of model C collects response messages from the OUT1 ports of model A and B;
4) it sends these messages back to the In1 port of model D.

The state synchronization mechanism works from bottom-to-top. In other words, a parent models execute the state synchronization mechanism after the completion of all of its submodels. As mentioned before, communication in SiMA is bidirectional and destination ports are aware of the source ports they are connected to. However, while updating the states, destination ports are not aware of the source ports that they are newly connected to. As can be observed in Algorithm 7, sources of destination ports are synchronized after the state updating procedure. In this way, SiMA prevents unnecessary queries, as well as duplicate messages. For example, in Figure 7 two couplings are added dynamically, from the OUT2 port of model C to the IN port of model D and the IN port of model D to the IN port of model E. In this example, the state synchronization process works as follows:

- From the IN port of model D to the IN port of model E:
  1) the IN port of model E sends a request to the IN port of model D;

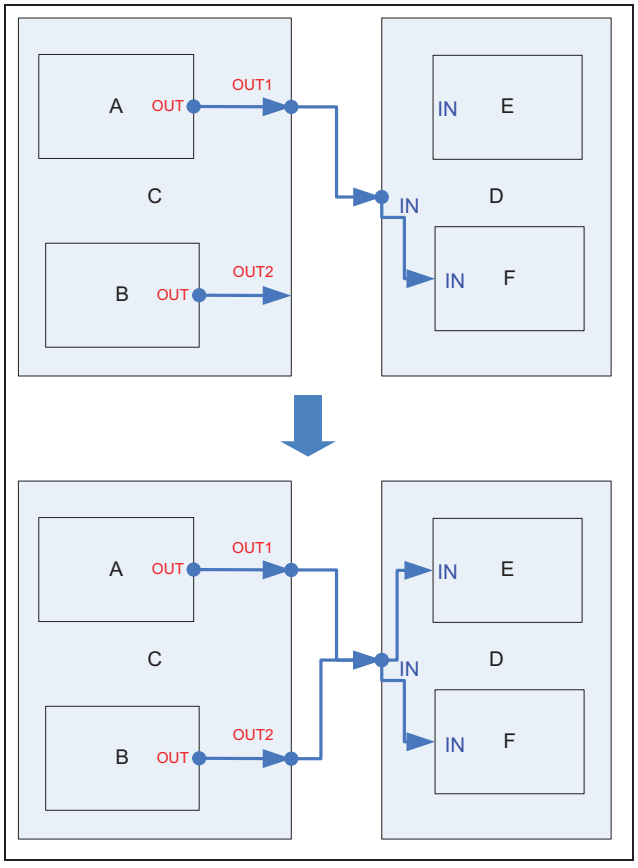

Figure 7. Dynamically adding couplings.

  2) the IN port of model D redirects this request to the OUT1 port of model C;
  3) the OUT1 port of model C redirects this request to the OUT port of model A;
  4) response messages of model A are sent to model E from the same path.
- from the OUT2 port of model C to the IN port of model D:
  1) the IN port of model D sends a request to the OUT2 port of model C;

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

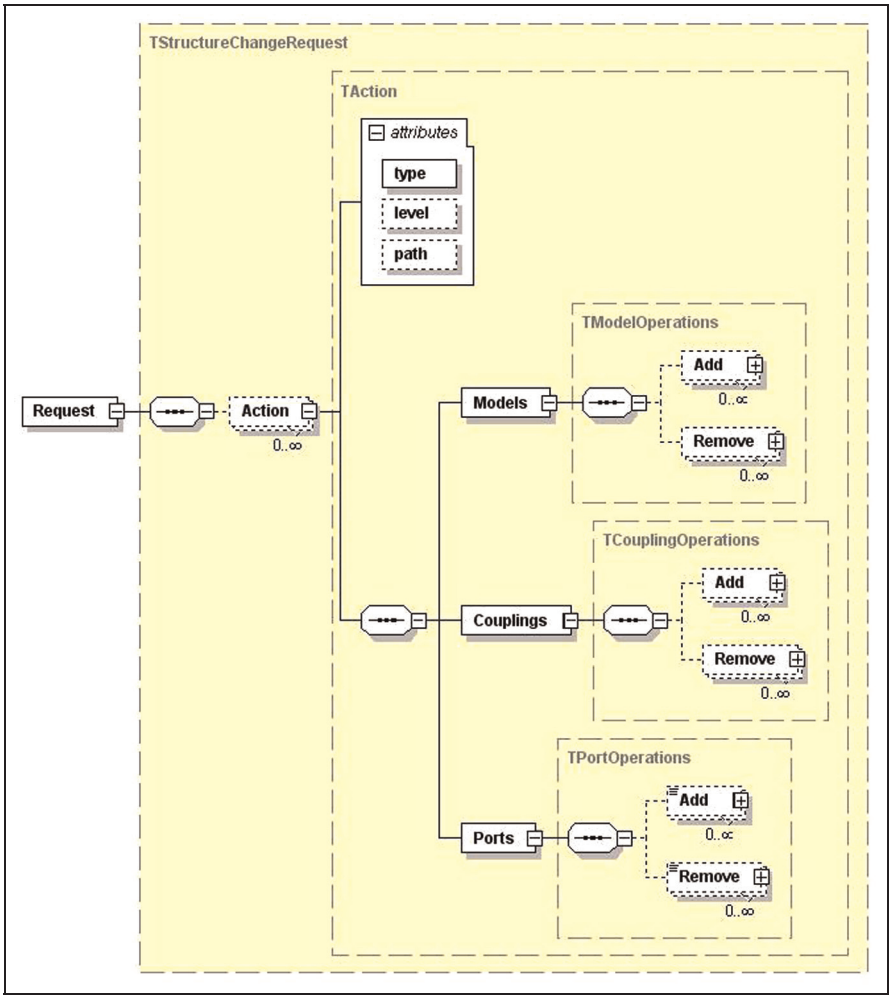

Figure 8. Change request message structure.

2) The OUT2 port of model C redirects this request to the OUT port of model B;
3) response messages are sent to the IN port of model D from the same path;
4) the IN port of model D sends a copy of these messages to the IN ports of both models E and F.

4.5 Change request message structure

Change requests in SiMA are defined as XML documents that are compatible to the XML Schema shown in Figure 8.

A change request consists of several actions and each action consists of attributes and structure change operations. Attributes of an action specify the destination model that will execute structure change operations defined in the action. An action has three attributes: type, level and path. Level and path are optional attributes and their data types are *int* and *string*, respectively. The type attribute of an action can have one of the following values: relative path, absolute path, relative level and absolute level. For example, if the type attribute is set to relative level and level attribute to one, then the action will be executed in the parent-coupled model. If the level attribute is set to two, then the action will be executed at the parent-coupled model of the parent-coupled model, and so on.

In dynamic SiMA, requests can be sent both from top-to-bottom and bottom-to-top. For both purposes, the same schema, introduced in this section, is used. The structural flexibility of this schema allows the requests to be broken into pieces for messages sent from top-to-bottom and allows the requests to be combined for messages sent from bottom-to-top.

4.6 Time management in dynamic SiMA

The time management diagram of a simulator in dynamic SiMA is illustrated in Figure 9. The parts that are clearly marked in the figure connote the dynamism extensions to

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

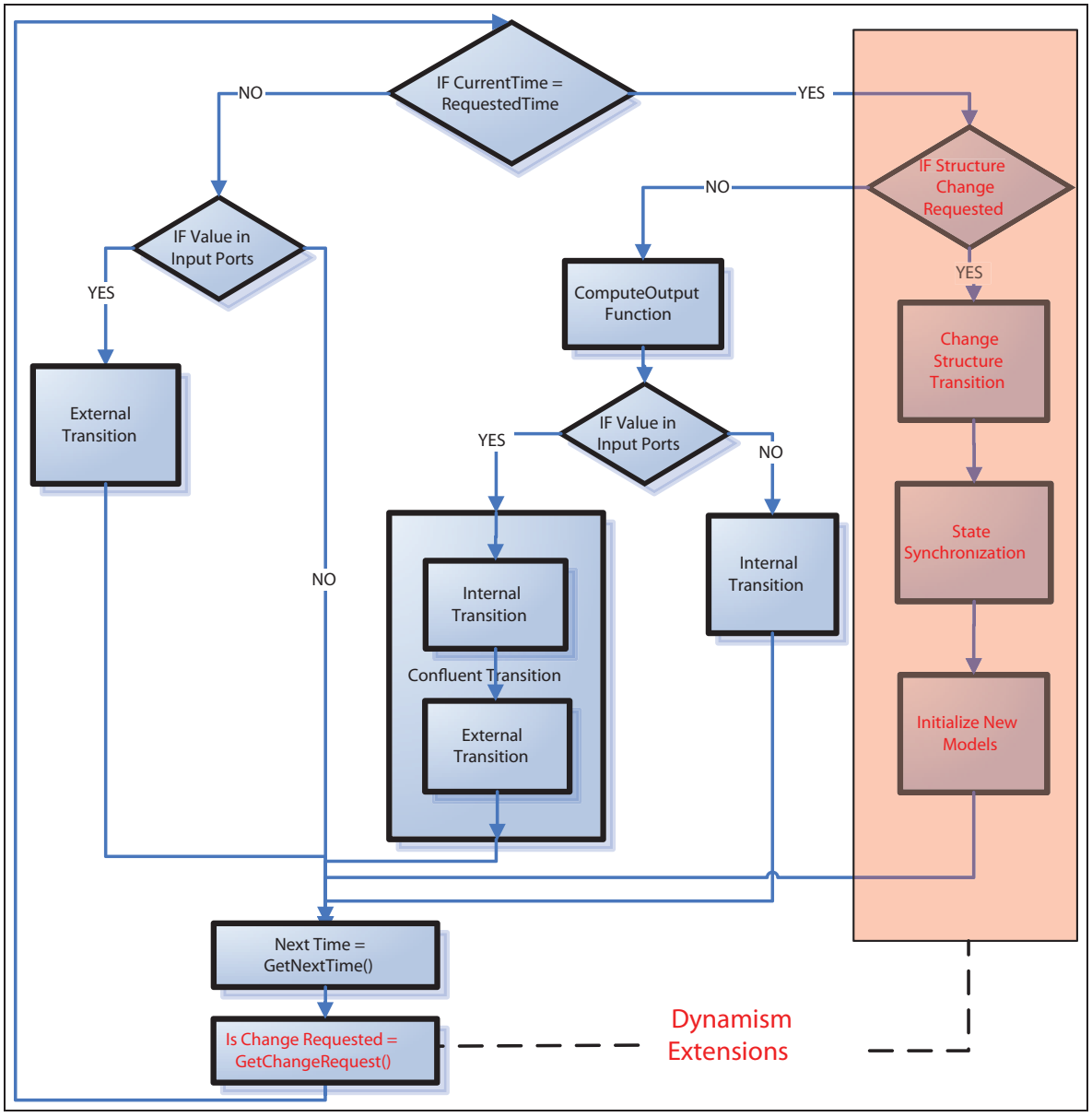

Figure 9. Time management in dynamic SiMA.

the time management flow of classical DEVS formalism. After each simulation cycle, the simulator checks whether there exists a structure change request in the next cycle. If such a request exists and if the current simulation time is equal to the time of the change request, then the required changes are applied. Changing the model structure in SiMA is handled in three steps.

- Changing the model structure.
- Synchronizing states (details are discussed in Section 4.4).
- Initialization of newly added models. Current times of newly added models are also advanced to the current simulation time.

After the structure is modified, the normal simulation cycle resumes with the updated model structure.

5. Case study and performance analysis

5.1 Wireless sensor network simulation as a case study

In this section, we present the details of our case study and report our performance measurements to illustrate the impact of SiMA dynamism extensions. To conduct the performance tests, a WSN simulation has been developed as a sample case. Models representing the sensors are implemented using both classic SiMA and dynamic SiMA frameworks and the performance findings are compared.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

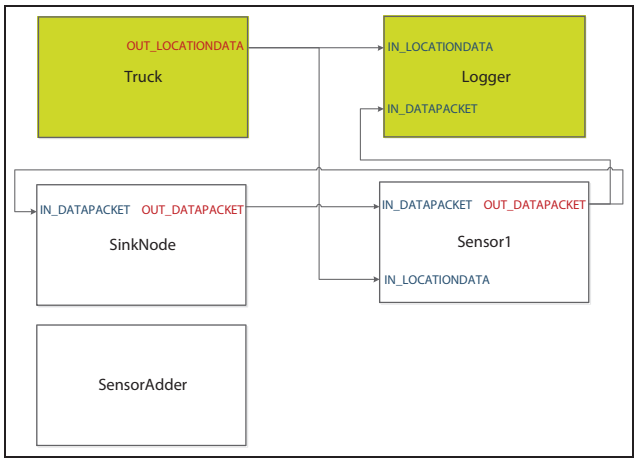

Figure 10. Wireless sensor network main model.

The domain knowledge required for designing a realistic WSN is obtained from the work presented by Iyengar and Chakrabarty[22] and Akyildiz et al.[23] A top-level visual representation of DEVS models developed to implement the proposed WSN is given in Figure 10. The WSN system consists of five components.

1) The **sensor** detects the movement of objects in the environment and can communicate with other sensors within its range. Each sensor is represented by a coupled model and consists of four subcomponents. These components and their inner relationships can be observed in Figure 11.
   - The **antenna** is used for communicating with the main sensor and other sensors within their range. As such, the antenna is the intermediate model between the outside world and the processor. Messages coming from the outside world are sent to the processor and messages received from the processor are sent to the other sensors. A routing protocol is also implemented in this model. For this case study, greedy forwarding, which is the simplest form of geographic routing, is used. Each node makes decisions according to the locations of its direct neighbors. Each sensor designates the neighbor that has the shortest distance to the sink model as its parent. More details about greedy forwarding algorithm can be found in Karp and Kung.[24]
   - The **sensing unit** is used for sensing the movement activities in the environment. When a sensing unit detects a movement it estimates a value between [0,1] representing the proximity of the detected object and sends a message containing this data to the processor.
   - The **processor** receives messages from both the antenna and the sensing unit. It creates and sends data messages to the antenna according to messages received from the sensing unit. When the processor receives a message from the antenna that is originated from the sink sensor, it sends an activation message to its sensing unit.
   - The **battery** is connected to the antenna, processor and sensing unit. The battery models power consumption. When the power of the battery runs out, the battery model informs other models with an event, upon which all models in the sensor change their states to 'Dead' phase.
2) The **main sensor** (or the **sink unit**) is the base unit that collects and evaluates all the information supplied by the sensor network. It can communicate with the sensors within its range but does not sense the environment. It sends activation messages to the sensors and waits for the response messages. Unlike other sensors, the main sensor does not contain a battery unit. The main sensor contains two subcomponents. These components and their relationships can be observed in Figure 12.
   - The **sink antenna** sends activation messages and listens to the incoming messages within a specific range. It redirects received messages to the sink processor.
   - The **sink processor** follows the truck's movement by processing the detection messages received from the sensors. During the simulation, it collects data messages received from the sensors and it analyzes these messages to determine the (time, location) pairs for the truck's movement. The analysis of the location of the truck at each time step is handled using the trilateration method discussed by Staras and Honickman.[25]
3) The **truck** has a predefined path that it follows during the course of simulation. The motion model does not include any dynamics so is very straightforward, simply changing the truck's position along the predefined path with a predefined velocity at each time step. The truck represents the detectable object for the sensors in the environment.
4) The **logger** acts as the transducer model and logs all the location and data packages, created by truck and sensors, respectively, into a file on the disk. This file can be used to analyze the simulation results.
5) The **sensor adder** exists only in the dynamic version of the simulation and it is used for adding sensors into the environment at run time.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

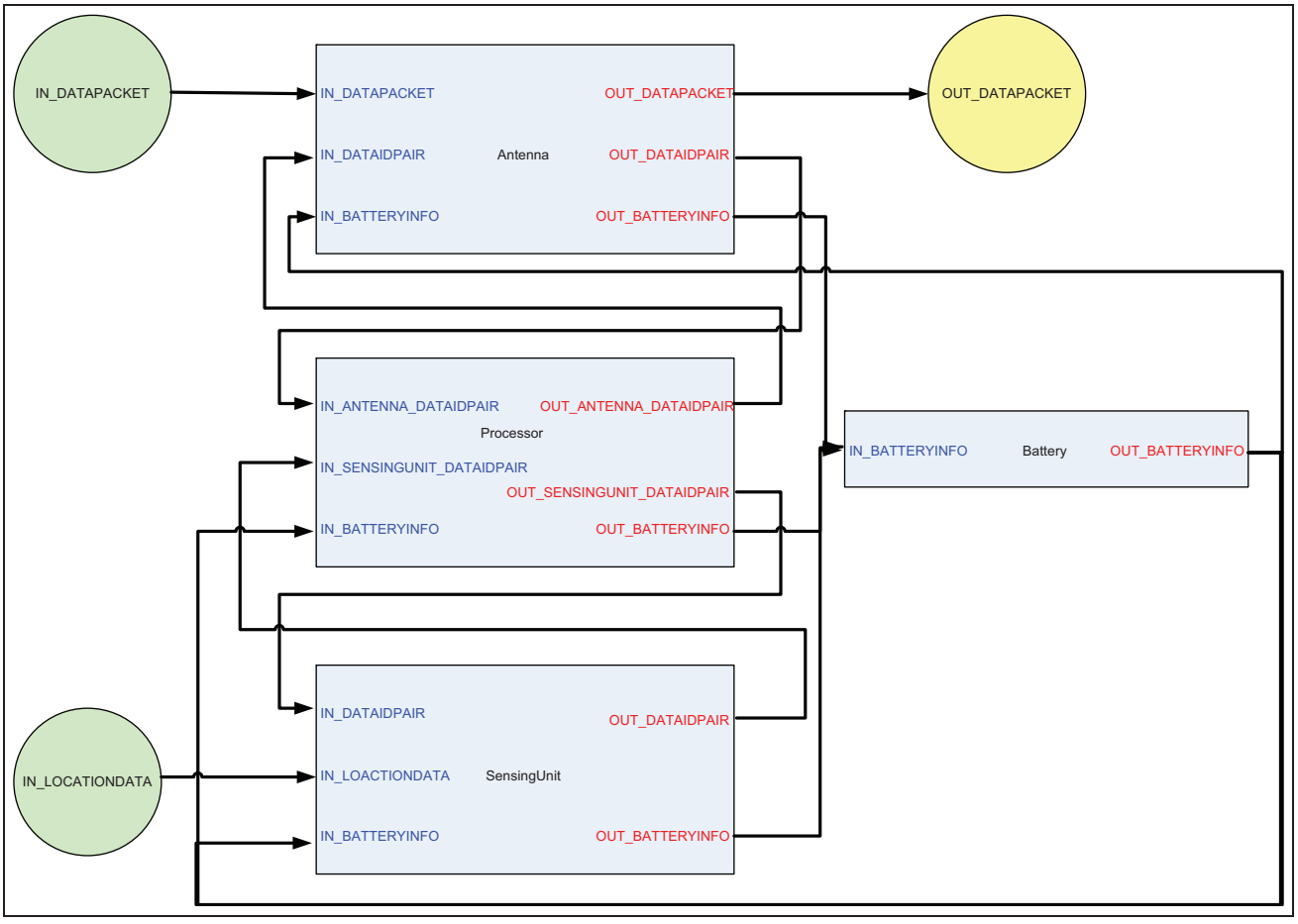

Figure 11. Single sensor model.

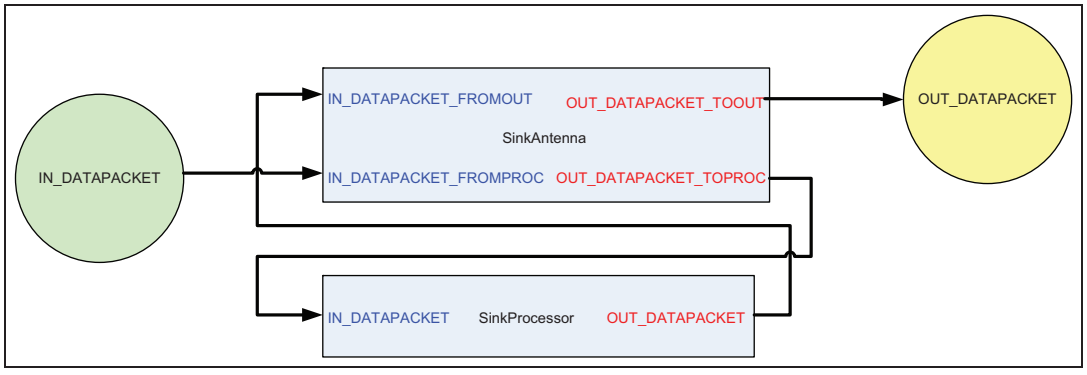

Figure 12. Main sensor model.

5.2 The simulation scenario

Our sample simulation scenario involves randomly distributing a number of wireless sensors into an area to construct a WSN system. A sink unit is positioned randomly within the area. Thus, in the initialization phase of the simulation, a sensor network with a random layout is deployed into an area and is ready for sensing intruder objects. When the simulation starts, a truck starts navigating the area where the sensor network was deployed. When the truck enters a sensor's sensing range, the sensor detects the truck's location and sends an accuracy value to its parent to be sent to the sink unit. At the end, the sink unit analyzes collected messages and determines the observed path of the truck. The simulation analyst is then

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

able to compare the actual path and the observed (or detected) path of the truck to determine the effectiveness of the WSN with the given characteristics (such as the sensor layout, sensing ranges, etc.).

In this case study, tests are performed according to the following scenario:

- a truck starts moving along a random path with a constant speed;
- the sink unit creates and broadcasts an activation message.;
- when a sensor receives the activation message, it broadcasts it to the maximum possible number of nodes;
- sensors that receive the activation message start collecting movement activity data from the environment via their sensing units;
- when a sensing unit receives movement activities from the truck, it sends a message to its processor;
- the processor creates data messages according to the collected data from the environment;
- when messages are ready, processors send them to their antenna to be sent to the sink unit.;
- the sink unit collects these data messages and resolves the current location of the truck.

5.3 WSN simulator application

To drive the tests and collect the metrics, a test application called the WSN Simulator Application is implemented. The application enables the user to observe the behavior of WSNs via a visually plausible graphical interface and to verify simulation results both through textual output and visual cues. It also provides a controlled experimentation environment which ensures that both static and dynamic cases are tested in a consistent way. Figure 13 illustrates the main window of the tool. We provide a quick scan through the capabilities of this tool to give some insight into our testing environment. The tool consists of the following five main panels.

- **Simulation parameters**: this panel contains the parameters that control the main characteristics of the simulation run. Values of these parameters are passed to the related simulation models to facilitate the initialization phase of these models. By changing parameters, such as sensor count, sensor adder frequency and simulation type (static or dynamic), different simulations can be executed.
- **Simulation manager**: this panel controls the simulation run. Before starting a simulation, simulation end conditions can be defined. While a simulation

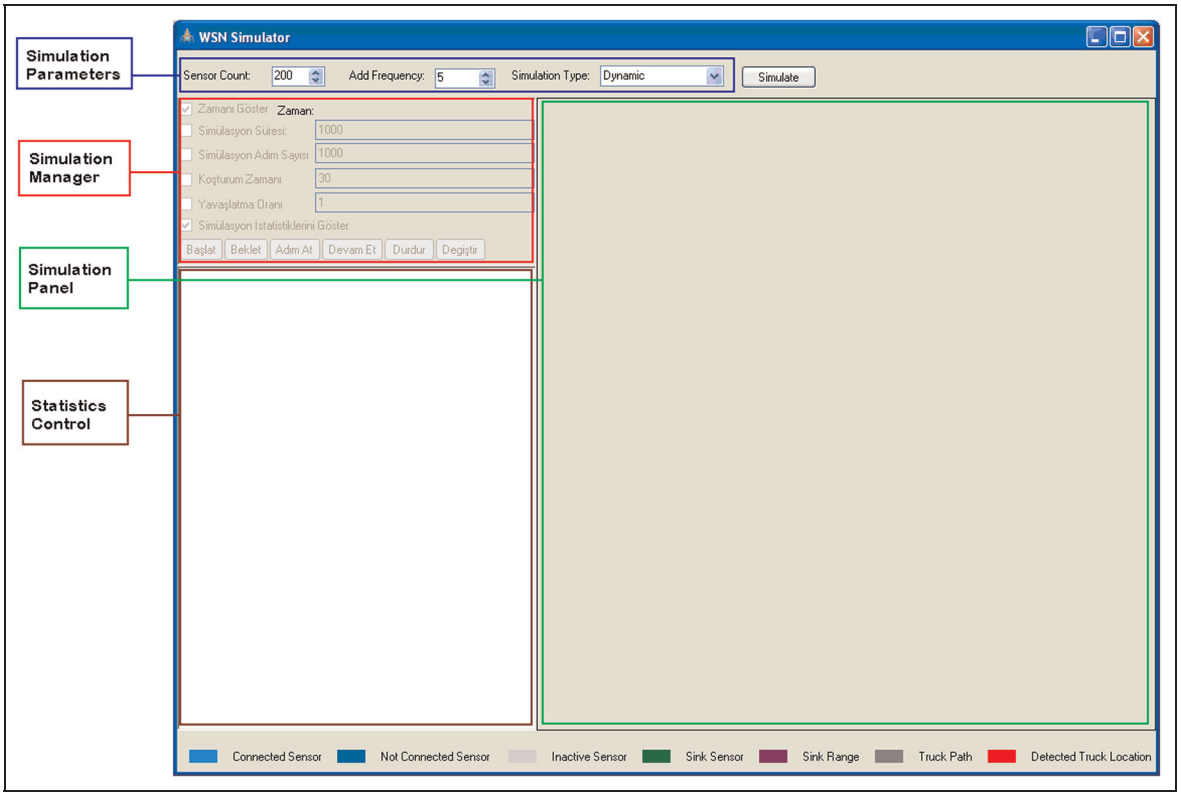

Figure 13. Wireless sensor network simulator tool.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

is running, it is possible to pause or stop the simulation. When the simulation is paused, allowed operations are iterating the simulation for one step, resuming execution, stopping the simulation and finally changing the model structure (for dynamic case). The command for changing the structure allowed here is a top-to-bottom structure change operation where upon user command (via the simulation manager panel), structure change requests are read from an XML document (for simplicity) and sent from root coordinator to the related subcomponents.

- **Simulation panel**: visual representation of the running simulation is shown in this section of the screen. A truck's path is shown with a polygon and sensors are represented by dots with different colors according to their states.
- **Statistics control**: statistics of completed simulations are shown in this section. By just looking into the visual representation of the simulation it may be difficult to make sure that both static and dynamic simulations have executed the same simulation. Hence, statistical data need to be used. In addition to the simulation execution time and simulation type, the number of times a truck is located and the number of sensors locating that truck are also shown as a part of statistical data and these are the main metrics we used for making sure that we are executing the same simulation with different approaches.

Figure 14 illustrates the simulator after executing a dynamic simulation. In the dynamic approach, after a sensor completes its execution, it is removed from the simulation structure. So, in Figure 14 only active sensors are visible. In the static case inactive sensors would also remain in the model structure and would be displayed in the simulation panel colored gray.

5.4 The performance tests

Performance measurements are carried out to observe the impact of the dynamism extensions. To achieve that the following steps are taken.

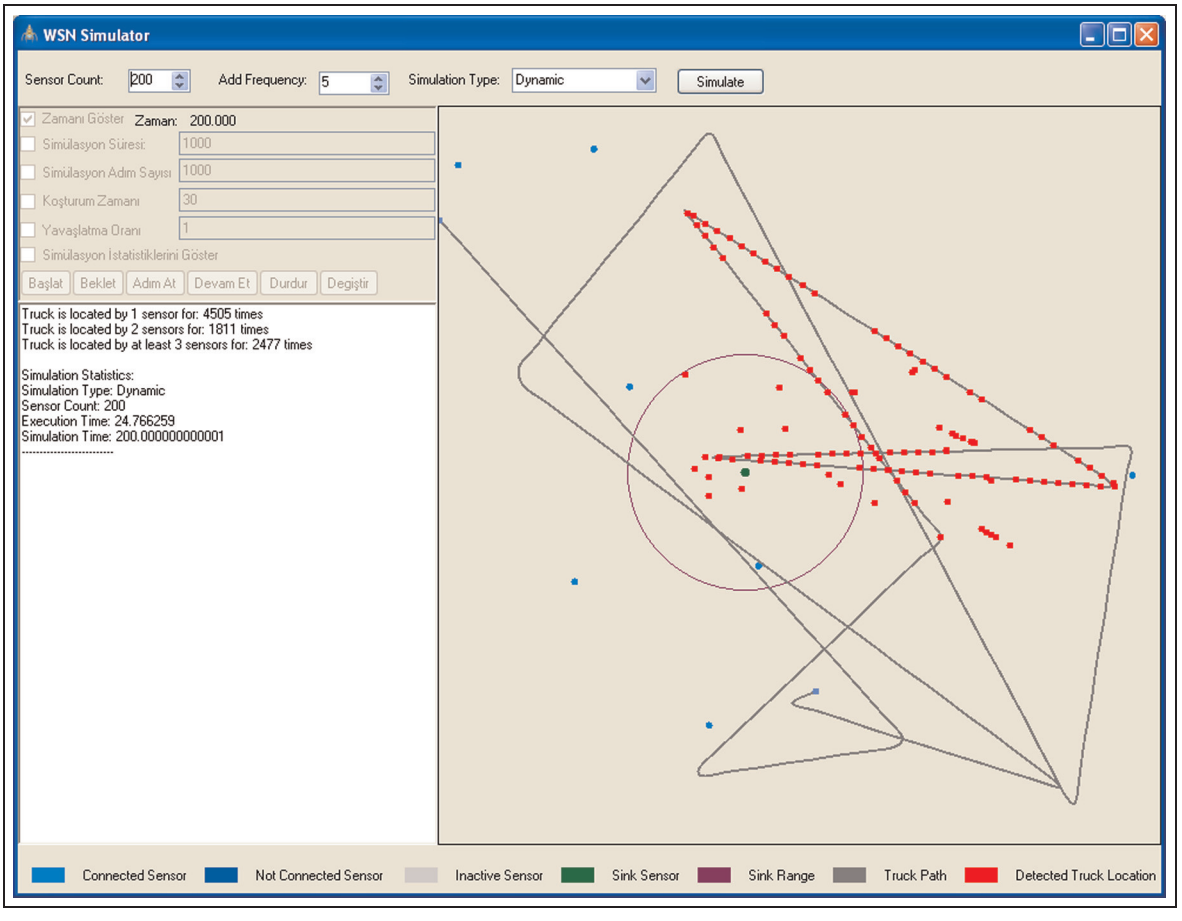

Figure 14. Wireless sensor network dynamic simulation.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

- Exactly the same simulation model set is, implemented using the basic SiMA framework, as well as the extended SiMA framework that includes dynamism support.
- The test tool presented above is used to run the simulation for both implementations.
- Two metrics are defined to be used for comparisons: (i) execution duration of simulations for varying sensor counts; and (ii) execution duration of simulation for varying truck step sizes. These metrics are collected for both dynamic and static (classical) cases.

Tests are performed in a Windows XP environment on a machine with the following hardware configuration: Intel(R) Core(TM)2 Quad central processing unit (CPU) Q9550 @2.83 GHz, 3.93 GB of random-access memory (RAM).

The rationale behind specifying the metrics given above and the method adopted for collecting them are presented below.

5.4.1 Testing criteria 1: sensor count. Execution times of simulations for varying sensor counts are our first metric. Sensor count is a good measure of structural dynamism in that it indicates the impact of dynamic model inclusion and removal as opposed to a static model coupling structure. Four of the simulation models (namely sink unit, logger, truck and sensor adder) have one instance only, whereas sensor models will be in varying numbers. For example, when there are 40 sensors in the simulation, the number of sensor models will be 90% of the whole model structure. In the dynamic version, these sensors are inserted into the simulation when they are activated and removed from the simulation when they are deactivated. However, in the static approach, all models are added to the simulation at the beginning and removed when the simulation is terminated. In Table 1 execution times of simulations for both static and dynamic approaches and performance improvement of the dynamic approach over the static approach can be observed. The improvement of the dynamic approach over the static approach is calculated as the percentage difference in the elapsed time. In the experiments the total simulation wall clock duration is set to be 200 seconds.

To aid the interpretation of the results shown in Table 1, Figures 15 and 16 are presented to provide a graphical representation. In Figure 15, execution times of the static and dynamic approaches are shown. As can be observed from the results, execution times of the static approach are proportional with the sensor model count, whereas execution times of the dynamic approach are almost proportional with the logarithm of the sensor model count. In Figure 16, performance gain of the dynamic approach over the static approach is graphically represented.

Table 1. Comparison of the two approaches according to sensor count (in seconds).

Sensor count	Static(s)	Dynamic(s)	Difference(s)	Gain (%)
1	0.30	0.27	0.03	10.00
3	0.72	0.64	0.08	11.11
5	2.55	2.37	0.18	7.06
7	4.50	4.20	0.30	6.67
9	6.79	6.29	0.50	7.36
11	8.97	8.25	0.72	8.03
13	12.24	11.02	1.22	9.97
15	15.78	14.07	1.70	10.77
17	19.02	16.55	2.46	12.93
19	22.37	18.98	3.39	15.15
21	25.89	21.06	4.83	18.66
23	30.14	23.79	6.35	21.07
25	33.95	26.13	7.81	23.00
27	38.20	28.87	9.34	24.45
29	42.42	30.93	11.49	27.09
31	46.61	32.70	13.90	29.82
33	49.69	34.02	15.67	31.54
35	53.48	35.33	18.15	33.94
37	56.39	36.14	20.25	35.91
39	59.22	36.73	22.49	37.98

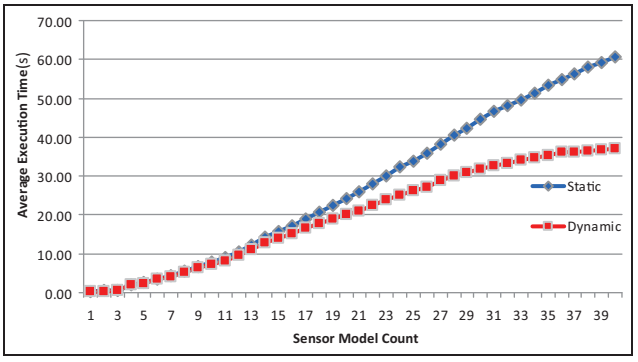

Figure 15. Execution time of simulations with respect to sensor count.

An important factor that influences the performance of the simulation is the total number of messages transferred between models. We have added the capability to each atomic model in the simulation to keep track of the number of received messages and log that statistic, in both static and dynamic cases. For the sample simulation a typical message size is 42 bytes. It can be observed that, in static simulations, the total number of messages circulating in the simulation is proportional with the square of the sensor count. On the other hand, in dynamic simulations, the rate of increase in the message count is much less with respect to the increase in sensor count. In Table 2, the number of messages for different sensor counts is shown for both static and dynamic simulations. The figures help us to reveal

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

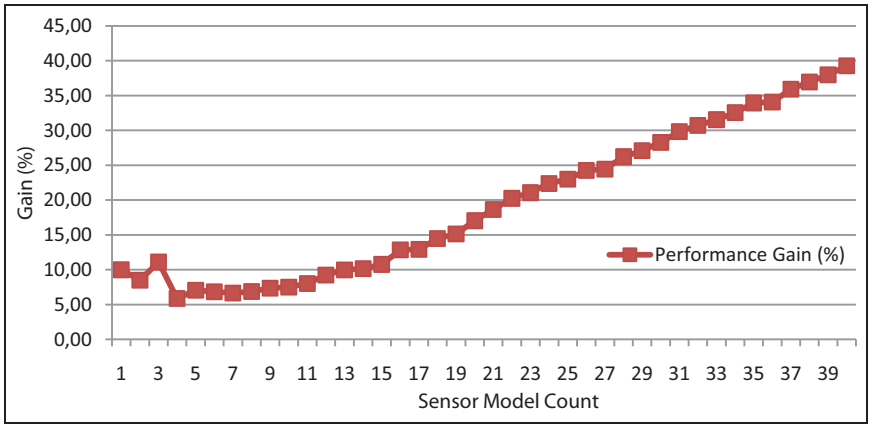

Figure 16. Performance enhancement of dynamic approach with respect to sensor count.

Table 2. Comparison of the two approaches according to total message count.

Sensor count	Message # in static	Message # in dynamic	Gain %
10	1,540,071	1,484,869	3.58
20	5,545,412	4,726,308	14.77
30	11,003,793	7,885,944	28.33
40	16,120,844	9,250,767	42.62

the reasons behind the performance improvement reported in Table 1.

5.4.2 Testing criteria 2: truck step size. The simulation step size of the truck model is another convenient parameter for increasing or decreasing the number of messages traveling along the couplings between the models, which is instrumental in measuring the effect of dynamic coupling management in reducing the message handling cost of the framework. The number of messages circulating between sensors is inversely proportional to the truck step size. Although the end result of this exercise is similar to the test conducted in Section 5.4.1 Testing criteria 1: - sensor count, this time the simulation resolution (and hence model resolution) is increased. Thus we aim to observe the behavior of the framework where the number of messages is increased in a more non-deterministic manner. This is because, in this test, the number of messages created depends on the number of sensors encountered along the path of the truck and smaller step sizes may cause the truck to be more sensitively sensed when passing along a more populated area, since the sensors are irregularly distributed over the area. Tests for measuring this criterion are executed for 10 sensors with varying truck step sizes for both static and dynamic approaches. The impact of truck step size on performances can be seen in Table 3.

Table 3. Comparison of the two approaches with respect.

Sensor count	Truck step size(s)	Static(s)	Dynamic(s)	Diff.(s)	Gain %
10	0.001	24.79	17.53	7.26	29.29
10	0.002	15.25	11.52	3.73	24.46
10	0.003	12.22	9.50	2.72	22.26
10	0.004	10.12	7.99	2.13	21.05
10	0.005	8.84	7.15	1.69	19.12
10	0.006	8.61	7.05	1.57	18.23
10	0.007	8.29	6.82	1.46	17.61
10	0.008	8.07	6.72	1.34	16.60
10	0.009	8.05	6.74	1.31	16.27
10	0.010	7.52	6.44	1.07	14.23

The Results of the tests are presented in Table 3 and are graphically illustrated in Figures 17 and 18. Examining Figure 17, we observe that increasing the step size (i.e. decreasing the resolution) improves the performance of both approaches. This is an expected behavior. When the resolution decreases (i.e. step size increases), the number of messages circulating between models decreases and therefore performance increases. On the other hand, Figure 18 signifies that performance degradation of the dynamic approach as the resolution increases is much less compared to that of the static approach. When the simulation resolution increases, the gap between the dynamic approach and static the approach becomes wider. This indicates that, for higher resolutions, the benefit of the dynamic approach is more evident.

5.4.3 Discussion of the results. A summary of the reasons for the reported findings can be enumerated as follows.

1) In classical DEVS formalism, after the initialization phase, simulation goes into a loop where in

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

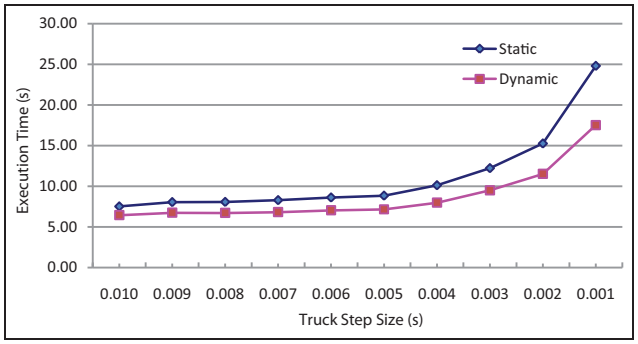

Figure 17. Execution time of simulations with respect to truck step size.

each cycle, all simulators execute their functions as mandated by the applicable DEVS simulation protocol. The number of simulators in classical DEVS is fixed (as the number of models is fixed) and does not change while the simulation is running. However, in the dynamic version, models can be inserted into the simulation whenever they are needed and they are removed when they complete their work. Therefore, the number of simulators in the dynamic version will always be less than or equal to the number of simulators in the classical version. As a consequence, the number of calls to simulators in each cycle of the simulation loop never exceeds that of the classical approach.

2) Even though incoming messages are not processed by inactive models, messages are nevertheless received by their ports and then they are discarded. Therefore, not only the existence of the inactive models, but also unnecessary couplings between those models, cause performance degradation during simulation execution. The DynDEVS approach eliminates such performance losses through its support for dynamic coupling management.

One particular work that focuses on the performance analysis of DSDEVS based on cellular space models is that of Sun and Hu.[26] Their findings are similar to ours in that they also report the important positive impact of dynamic structure modeling due to reduced memory requirements, faster simulation initialization periods and improved efficiency of the simulation engine because of the small proportion of active models (cell spaces in their case). However, they also report that the overhead of dynamically adding and removing models at run time can become significant for simulations that have a high proportion of active models. They note that in such cases the advantage of not constructing all the models at the start of the simulation diminishes. Intuitively, it seems to be evident that the benefit of dynamic structure modeling is likely to be pronounced when the proportion of the active models with respect to all required models in the overall simulation remains below a certain threshold during the course of simulation. In fact, in our case, due to the state synchronization mechanism introduced, this overhead may even become more dominant for extreme cases where the proportion of the active models remains high and the state synchronization cost is particularly significant.

It is worth noting that the state synchronization mechanism introduced as part of SiMA-DEVS dynamism extensions is fully utilized in the case study. For instance, in the sample WSN simulation, sensors start sensing the movement of the truck in the environment upon receiving activation messages from the main sensor, indicating that it has started execution. The main sensor sends this message only once after the initialization phase. When a new model is included into the simulation dynamically, it needs to know whether the main sensor is executing so that it can start sensing the environment. In order to enable newly added sensors to acquire the current state of the simulation, the state synchronization mechanism is employed.

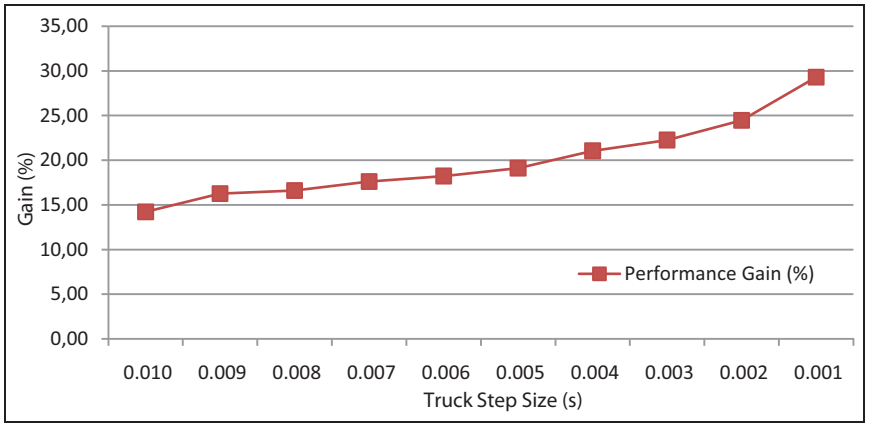

Figure 18. Performance enhancement of dynamic approach in relation to truck step size.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

6. Conclusion and future work

In this paper, we have introduced our approach to implement variable structure support for dynamic and adaptive simulation environments. We summarized the fundamental properties of our modeling and simulation framework, SiMA, and its variable structure extensions, with references to similar approaches in the literature.

In particular we note the following.

1) We have formally presented the extensions to SiMA-DEVS to support structural dynamism. We reiterate that our approach to add dynamism to our basic DEVS model is derived from to that of DynDEVS. However, we do not have the limitations that DynDEVS has. Unlike DynDEVS, our approach allows dynamic port management and allows atomic models to initiate changes other than changing themselves only. One particular contribution we offer is the formal specification and systematic framework support for post-structural-change state synchronization among models with related couplings. This operation works in the opposite direction of the normal message flow and enables newly added models and newly added couplings to acquire the current state of the simulation. This feature is used and tested in the sample simulations.
2) We have extended our existing implementation of SiMA-DEVS by incorporating the mechanisms required to implement the structural dynamism. This extended version of SiMA, namely dynamic SiMA, is used during the experiments.
3) In order to test our approach, we have developed a sample WSN simulator that uses dynamic SiMA. Since dynamic SiMA is an extended version of basic SiMA, the simulation engine was able to execute both static and dynamic simulations. Using this simulator, we executed several scenarios with different parameters and measured the performance according to two different metrics: (i) the sensor count in the simulation; and (ii) the simulation step size of the target model. As a result, using the sensor count metric, we observed in Section 5.4 that as the model structure complexity increased and the performance gain of the dynamic approach over the static approach also increased. According to our second metric, we observed that, as the truck step size (hence the resolution of the simulation) increased, dynamic SiMA improved the overall performance by 40%. We argue that these results indicate the utility of dynamism support in improving the performance of the simulations.

As a concluding remark, we would like to provide a comparative summary of our approach with the two most closely related approaches, namely DSDEVS and DynDEVS. Our approach is mostly aligned with DynDEVS with non-disruptive extensions to the overall semantics of the basic DynDEVS formalism. A comparative summary of the most important features of these approaches can be found in Table 4.

Table 4. Feature comparison of approaches.

	DSDEVS	DynDEVS	Dynamic SiMA
Adding/removing model	Yes	Yes	Yes
Adding/removing coupling	Yes	Yes	Yes
Adding/removing port	Yes	No	Yes
State synchronization	No	No	Yes

Funding

This work was partially supported by the research and development (R&D) office of the Turkish Ministry of Defense.

References

1. Uhrmacher AM. Dynamic structures in modeling and simulation: a reflective approach. ACM Transactions on Modeling and Computer Simulation; April 2001; 11 (2): 206–232
2. Kara A, Bozagac CD and Alpdemir MN. SiMA: A DEVS Based Hierarchical and Modular Modelling and Simulation Framework. 2nd National Defensive Applications Modelling and Simulation Conference, Ankara, Turkey, 2007
3. Kara A, Deniz F, Bozagac CD and Alpdemir MN. Simulation Modeling Architecture (SiMA), A DEVS Based Modelling and Simulation Framework. In Proceedings of Summer Computer Simulation Conference (SCSC'09), İstanbul, Turkey, July 2009, pp. 315–321.
4. Baati L, Frydman C and Giambiasi N. LSIS_DME M&S environment extended by dynamic hierarchical structure DEVS modeling approach. In: *Proceedings of the 2007 Spring Simulation Multiconference*, San Diego, CA, 2007, pp.227–234.
5. Barros FJ. Dynamic structure discrete event system specification: anew formalism for dynamic structure modeling and simulation. In: *Proceedings of the 1995 Winter Simulation Conference*, 1995, pp.781–785.
6. Lee K, Choi K, Kim J and Vansteenkiste GC. A methodology for variable structure system specification: formalism, framework, and its application to ATM-based network system. *ETRI J* 1997; 18: 245–264.
7. Pawletta T and Pawletta S. A DEVS-based simulation approach for structure variable hybrid systems using high accuracy integration methods. In: *Proceedings of the Conference on Conceptual Modeling and Simulation, Part of Mediterranean Modelling Multiconference*, Genoa, Italy, 2004, pp.368–373.
8. Shang H and Wainer GA. A flexible dynamic structure DEVS algorithm towards real-time systems. In: *Proceedings of the 2007 summer computer simulation conference*, San Diego, CA, 2007, pp.339–345.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015

9. Hu X, Zeigler BP and Mittal S. Variable structure in DEVS component-based modeling and simulation. *Simulation* 2005; 81: 91–102.
10. Zeigler BP, Kim TG and Praehofer H. Theory of Modeling and Simulation (2nd ed.). Academic Press, Inc., Orlando, FL, USA, 2000.
11. Zeigler BP. Theory of Modeling and Simulation. John Wiley, New York, 1976.
12. Barros FJ, Zeigler BP and Fishwick PA. Multimodels and dynamic structure models: an integration of DSDE/DEVS and OOPM. In: *Proceedings of the Winter Simulation Conference*, 1998, pp.413–420.
13. Cardelli L. Type Systems. CRC Handbook of Computer Science and Engineering, 97:1-97:41, Boca Raton, Florida: Chapman&Hall/CRC Press, 2004
14. Fujimoto RM and Weatherly RM. Time management in the DoD high level architecture. In: *Proceedings of the Workshop on Parallel and Distributed Simulation*, Institute of Electrical and Electronics Engineers, Piscataway, 1996, pp.60–67.
15. Hwang MH. Identifying equivalence of DEVSs: language approach. In: *Proceedings of Summer Computer Simulation Conference*, Montreal, Canada, 2003, pp.319–324.
16. Hwang MH. Equivalence and minimization of output augmented DEVS. In: *Proceedings of the IEEE Conference on System, Man, and Cybernetics*, Washington, DC, 2003, pp.409–414.
17. Hwang, MH and Cho SK. Timed behavior analysis of schedule preserved DEVS. In: *Proceedings of Summer Computer Simulation Conference*, San Jose, CA, 2004, pp.173–178.
18. Barros FJ. The dynamic structure discrete event system specification formalism. *Trans Soc Comput Simulat Int* 1996; 13: 35–46.
19. Barros FJ. Modeling formalisms for dynamic structure systems. *ACM Trans Model Comput Simulat* 1997; 7: 501–515.
20. Barros FJ. Abstract simulators for the DSDE formalism. In: *Proceedings of the 30th Conference on Winter Simulation*, Los Alamitos, CA, 1998, pp.407–412.
21. Himmelspach J and Uhrmacher AM. Processing dynamic PDEVS models. In: *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 2004, pp.329–336.
22. Qi H, Iyengar SS and Chakrabarty K. Distributed sensor networks–a review of recent research. *J Franklin Inst* 2001; 338: 655–668.
23. Akyildiz IF, Su W, Sankarasubramaniam Y and Cayirci E. Wireless sensor networks: a survey. *Comput Networks* 2002; 38: 393–422.
24. Karp B and Kung HT. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In: *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking*, Boston, MA, USA, August 2000, pp. 243–254
25. Staras H and Honickman SN. The accuracy of vehicle location by trilateration in a dense urban environment. *IEEE Trans Veh Technol* 1972; 21: 38–43.
26. Sun Y and Hu X. Performance measurement of dynamic structure DEVS for large scale cellular space models. *Simulation* 2009; 85: 335–351.

Author Biographies

Fatih Deniz is a PhD student in the Department of Computer Engineering at the Middle East Technical University (METU), Ankara, Turkey. He received his BSc degree from Bilkent University, Ankara, Turkey in 2007 and his MSc (2010) degree from the Department of Computer Engineering of the METU. His current research interests include model-driven engineering and variable structure models.

M. Nedim Alpdemir received his MSc (1996) in Advanced Computer Science and PhD (2000) in Component-Based Simulation Environments from the Department of Computer Science, University of Manchester, UK. He worked as a research associate, and later as a research fellow in the Information Management Group (IMG) at the Department Computer Science of the University of Manchester, UK, until 2005. Currently he is the head of the Software Infrastructures Group and supervises the Simulation Software Frameworks team at TUBITAK UEKAE ILTAREN, Ankara, Turkey.

Ahmet Kara is a chief researcher at TUBITAK UEKAE ILTAREN. He has been involved in the design and implementation of modeling and simulation architectures. He received his BSc (2003) and MSc (2006) degrees from Bilkent University, Ankara, Turkey. He is currently a PhD student in the Department of Computer Engineering of the METU. His current research interests include multi-resolution modeling in simulation systems.

Halit Oğuztüzün is an associate professor in the Department of Computer Engineering at the METU, Ankara, Turkey. He obtained his BS and MS degrees from the METU in 1982 and 1984, and PhD from University of Iowa, Iowa City, IA, USA, in 1991. His current research interests include distributed simulation and model-driven development. He participates in the activities of the Modelling and Simulation Research Center of the METU.

Downloaded from sim.sagepub.com at INRIA on July 7, 2015