Variable Structure and Dynamism Extensions to SiMA, A DEVS Based Modeling and Simulation Framework

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

[1]TUBITAK UEKAE İLTAREN
06800 Ümitköy
Ankara, TURKEY
{fatihd, ahmetk, nedima}@iltaren.tubitak.gov.tr

[2]Dept. of Computer Engineering
Middle East Technical University
06530 Ankara, TURKEY
oguztuzn@ceng.metu.edu.tr

Keywords: DEVS, simulation frameworks, strongly-typed environments, variable structure models

Abstract

In this paper we take a particular stand to the problem of dynamism support in simulation environments by adopting DEVS based modeling and simulation approach and by building upon our previous work on SiMA, a DEVS-based simulation framework developed at TUBITAK UEKAE. Our contribution to the work in this field is two fold: 1 – we have implemented a specialized form of basic DEVS formalism via SiMA (Simulation Modeling Architecture) 2- We have extended SiMA with dynamism support by building upon our specialized basic DEVS formalism with dynamism extensions that are comparable (but not equivalent) to those given in dynDEVS. Our approach conforms to both dynDEVS and dynNDEVS as the underlying formal specification, with some non-disruptive extensions to the original formal semantics. One particular contribution we offer 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.

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 (as indicated in [Uhrmacher 2001]), it may stem from the modeling and simulation methodology of the analyst or it may be due to the way system modelers approach to the modeling of inherent behavioral complexity of their models. A good example to 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. Allowing modifications to model structures and to internal functional specifications while the simulation is running 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 paper we take a particular stand to the problem of dynamism support in simulation environments by adopting DEVS based modeling and simulation approach and by building upon our previous work on SiMA [Kara et al. 2007], a DEVS-based simulation framework developed at TUBITAK UEKAE. We note that other approaches to dynamism are already proposed in the relevant literature [Barros 1995; Uhrmacher 2001; Zeigler et al. 1991]. We observe that three distinct categories of change are discussed in those existing approaches: 1 – A change in the overall compositional state of models, 2 - A change in the connectivity relationships (coupling) among the models, 3 – A change in internal functional behavior of the model.

We also note that there are two main formal approaches to the variable structure models in DEVS environment, which are defined by extending the classical DEVS formalism. The first one is DSDE (Dynamic Structure DEVS), introduced by F.J. Barros [Barros 1995]. The second one is dynDEVS, introduced by Uhrmacher [Uhrmacher 2001]. A brief introduction to both of these approaches is given in Section 2 of this paper. In addition to these formal extensions, there are approaches which adopt existing formal specifications but contribute through different routes. For instance [Shang and Wainer 2007] extend their existing simulation engine by adopting a combination of DSDE and dynDEVS. Similarly [Hu et al. 2005] take a software engineering oriented stand and propose a component-based simulation environment. Our contribution to the work in this field is two fold: 1 – we have implemented a specialized form of basic DEVS

formalism via SiMA (Simulation Modeling Architecture) (see Section 3 for more details) 2- We have extended SiMA with dynamism support building upon our specialized basic DEVS formalism with dynamism extensions that are comparable (but not equivalent) to those given in dynDEVS

The rest of this paper is organized as follows: Section 2 gives a summary of the relevant background work Section 3 provides a detailed discussion of our approach; Section 4 provides the conclusions and future work

2. BACKGROUND

The Discrete Event System Specification (DEVS) is a formalism introduced by Zeigler (1976) to describe discrete event systems. An atomic model, say M, in classical DEVS formalism consists of a set of input events, a state set, a set of output events, an internal, external, and confluent transition function, an output and time advance function, defined formally as follows [Zeigler 1976; Zeigler et.al., 1998]: $M =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, ta >$

where X is the set of input events; S is the set of states; Y is the set of output events; δint :S→S is the internal transition function; δext : QxX→S is the external transition function, where $Q = \{(s, e) \mid s \in S, 0 \le e \le ta(s)\}$ is total state set, e is elapsed time since last transition; λ : S→Y is the output function; ta : $S \to R^{+}_{0,\infty}$ is the time advance function. Complete description of DEVS semantics can be found in [Zeigler 1976; Zeigler et.al., 1998].

The Dynamic Structure Discrete Event System Specification (DSDE) is introduced in [Barros 1997] and allows the specification of dynamic structure networks of discrete event systems. A DSDE network model is described as follows [Barros 1996]:

$DSDEVN =< X_{\Delta}, Y_{\Delta}, \chi, M_{\chi} >$ where Δ is network name; χ is the name of DSDE network executive; M_{χ} is the model of χ; X_{Δ} is the set of input events; Y_{Δ} is the set of output events. M_{χ}, the model of the network executive χ, is a basic DSDE model and defined as:

$M_{\chi} =< X_{\chi}, S_{\chi}, Y_{\chi}, \delta_{\text{int}_{\chi}}, \delta_{ext_{\chi}}, \lambda_{\chi}, ta_{\chi} >$.

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:

$s_{\chi} = (D_{\chi}, \{M_i^{\chi}\}, \{I_i^{\chi}\}, \{Z_{i,j}^{\chi}\}, SELECT^{\chi}, V^{\chi})$ where

D_{χ} is the set of component names; M_i^{χ} is the model of component i, for i ∈ D^{χ}; I_i^{χ} is the set of component influencers of i, ∀ i ∈ D^{χ} ∪ {χ,Δ}; $Z_{i,j}^{\chi}$ is the i-to-j output to input function, ∀ j ∈ I_i^{χ}; $SELECT^{\chi}$ is the select function; V^{χ} represents other state variables of the network executive.

In DSDE, only the network executive can make structural changes and any change made in one of these 5-tuples $(D_{\chi}, \{M_i^{\chi}\}, \{I_i^{\chi}\}, \{Z_{i,j}^{\chi}\}, SELECT^{\chi})$ will be automatically reflected to the structure of the network model. A detailed explanation of DSDE formalism is found in [Barros 1997], and abstract simulators necessary to simulate DSDE models is found in [Barros 1998].

Unlike DSDE, dynDEVS formalism [Uhrmacher 2001] does not introduce a specific 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. There are two types of models defined in dynDEVS formalism. These are dynDEVS (atomic) and dynNDEVS (coupled) models. Atomic models are defined as follows:

$dynDEVS = df < X, Y, \text{m}_{\text{init}}, M(\text{m}_{\text{init}}) >$ where X,Y are structured sets of inputs and outputs; $\text{m}_{\text{init}} \in M(\text{m}_{\text{init}})$ is the initial model; $M(\text{m}_{\text{init}})$ is the least set having the structure $\{< S, \text{s}_{init}, \delta_{\text{int}}, \delta_{ext}, \lambda, ta, \rho_{\alpha} >\}$ where S is the set of states; $\text{s}_{init} \in S$ is the initial state; $\delta_{\text{int}}, \delta_{ext}, \lambda, ta$ are the same functions as in classical DEVS formalism; $\rho_{\alpha} : S \to M(m_{init})$ is the model transition function. Coupled models, which are composition of components and the links between these components, are described in dynDEVS formalism as follows:

$dynNDEVS = df < X, Y, n_{\text{init}}, N(n_{\text{init}}) >$ where X,Y are structured sets of inputs and outputs; $n_{\text{init}} \in N(n_{\text{init}})$ is the start configuration; $N(n_{\text{init}})$ is the least set having the structure $\{< D, \rho_{N}, \{dynDEVS_i\}, I_i, Z_{i,j}, Select >\}$ 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 dynamic DEVS models 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; $Select$ is the tie-breaking function. More details about dynDEVS formalism are found in [Uhrmacher 2001; Himmeelspach and Uhrmacher 2004].

3. OUR APPROACH

Our modeling and simulation framework SiMA (Simulation Modeling Architecture) 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 which:

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 semantic constraints on the port definitions;

2 – Introduces a new transition function to account for model interactions involving state inquiries with possible algebraic transformations (but no state change), without simulation time advance. In this respect we extend the basic DEVS formalism. This is similar to the notion of zero-lookahead in HLA [Fujimoto 1996] from a time-management point of view.

Our SiMA- DEVS formalism is given below:

$$SiMA-DEVS =< X, S, Y, \delta_{\text{int}}, \delta_{ext}, \lambda, ta, \delta_{df} >$$

$$X : Set\ of\ input\ values\ arriving\ from\ set\ of\ input\ ports, P_{in}$$

$$Y : Set\ of\ output\ values\ sent\ from\ set\ of\ output\ ports, P_{out}$$

$$P_{in}, P_{out} : Set\ of\ input\ and\ output\ ports\ such\ that:$$

$$P_{in} = \{(\tau, I_x) \mid \Gamma \mapsto \tau \wedge I_x \subseteq X \wedge \forall x \in I_x, \tau \mapsto x\},$$

$$P_{out} = \{(\rho, O_y) \mid \Gamma \mapsto \rho \wedge O_y \subseteq Y \wedge \forall y \in O_y, \rho \mapsto y\},$$

$$\Gamma : XMLSchema\ type\ system,$$

$$\tau, \rho : data\ types\ valid\ wrt\ XMLSchema\ type\ system,$$

$$\delta_{df} : PDFT_{in} \in P_{in} \times S \to P'_{out} \subseteq P_{out}$$

Note that the set of input ports P_{in}, 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, τ, is a data type conforming to XMLSchema type system (denoted with Γ) and the second element of the pair (I_x) denotes the set of input data values flowing through that port, where each element of the value set conforms to data type τ. Similar semantics apply to output ports, too. Thus, we make *strong typing* and *type-system dependency* of the ports explicit in the formal model. Although introducing a run-time oriented property into the formal model may seem unusual, we argue that there are a number of merits in doing so:

1. We introduce a type discipline to the definition of the externally visible model interfaces (i.e. ports) leading to an information model for the overall system being modeled (coherency in modeling level information space), as well as for the simulation environment (consistency and robustness in run-time-level data space).
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 we have successfully implemented our Model-Driven simulation construction pipeline for SiMA, via a number of tools such as a code generator, a model builder and a model linker.
3. We reduce the gap between modeling level logical composability constraints and run-time level pluggability constraints, thus forcing all implementations of our specialized DEVS model to respect our type-system compatibility and to offer a strongly typed environment.

Note also that, in addition, we introduce a new transition 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 models can share parts of their state, use state variables to compute derived values instantly within the same simulation time step. As stated earlier, this is similar to the notion of zero-lookahead found in HLA [Fujimoto 1996]. 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 inqury via existing couplings, then re-adjusting the time advance to go back to normal simulation cycle. Although this is possible, we argue that by introducing a transition function and a specific port type which is tied (through run time constraints imposed by the framework) to that particular transition function we gain several advantages:

1. The models can communicate and share 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 (compile time and run-time checks are carried out) the framework is able to apply application independent loop-breaking logic at the ports to prevent algebraic loops, thereby ensuring model legitimacy.

Further, our approach to add dynamism to our basic DEVS model is similar 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 trivial extensions which are given below:

1. We state that structured sets of inputs and outputs *X and Y* are defined in conformance to our strongly typed-port definitions where the formal definitions for P_{in}, P_{out} apply; $M(\text{m}_{\text{init}})$ is the least set having the structure $\{< S, \text{s}_{init}, \delta_{\text{int}}, \delta_{ext}, \lambda, ta, \rho_{\alpha}, \delta_{df} >\}$

where S, s_{init}, $\delta_{\text{int}}, \delta_{ext}, \lambda, ta$ and ρ_{α} are the same as in dynDEVS formalism; δ_{df} is the additional transition function defined in SiMA-DEVS. Thus, we ensure that the meaning of a model in dynDEVS is firmly aligned with that of SiMA-DEVS, while maintaining the top-level semantics of the dynDEVS definition. Note that our SiMA-DEVS extensions are non-disruptive to the overall semantics of the basic dynDEVS formalism.

2. We introduce a state synchronization mechanism between networks of connected models, to be performed at the end of a structural change phase, in case a model wants to update the values of such 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 stayed unchanged during pre-change simulation period. This mechanism is instrumental in cases where a model A initializes some of its state variables at the beginning of simulation but does not receive updates for those variables until some influencer model B goes through a structural change that causes those variables to be updated; or in cases where a new model B is added which introduces a new coupling influencing one of the input ports of model A. 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 which might potentially lead to significant errors in the behavior of the overall simulation, especially if the simulation application is developed for an engineering analysis requiring a high level 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

We now set out to describe the principles that govern the run-time algorithms of the SiMA simulation engine when it manages structural change. In SiMA, like dynDEVS, atomic models are responsible for initiating structural changes. There is no a dedicated controller model that supervises over atomic models as described in DSDE formalism. 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, and 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 the model addition may require the aggregation of state variables from many different models, or even it may be a user-initiated request which is not necessarily captured by a single model. In current implementations of dynDEVS namely AgedDEVS and JAMES, the atomic models are assumed to have access to a knowledge base from where they can collect the necessary information to decide for new model additions. Although this approach seems quite reasonable for agent-oriented implementations, it introduces a dependency to a specific architectural and behavioral semantics for simulation applications, which we are inclined to avoid. Therefore, in our approach:

- An atomic model may add/remove models or couplings to its parent coupled model. But these operations are restricted to its bounding coupled model, thus an atomic model cannot modify structure outside its coupled model.
- If an atomic model requires a structural change out of its parent coupled model it informs the 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 non-ambiguously aggregated, since our type-safe model composition semantics enables the resolution of any potential overlapping requests.
- An application that is running the simulation may require structural changes, too. This request is sent to the root coordinator to be executed over the model structure recursively. 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.
    - 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.

3.1. Operations on Model Structures

There are four types of structural change operations defined in SiMA: Adding a model, removing a model, adding a coupling and removing a coupling.

- *Removing a model:* This operation consists of two steps: Removing all the connections from/to the model. Removing the model.
- *Adding a model:* This operation consists of three steps:
  1. Adding a model to the parent coupled model.
  2. Calling 'init ()' function of the newly added model.
  3. Calling 'AdvanceTime(CurrentTime)' function for synchronization.
- *Removing a coupling:* The specified coupling is removed.
- *Adding a coupling:* This is also a critical operation in our case. After adding a coupling, a process for synchronizing current states of newly connected models is executed. For achieving this, a querying mechanism between connected ports is implemented that operates in the opposite direction of the normal message flow. An input port creates a query and sends this query to the newly connected ports. An answer to this query is generated and sent to the requesting port. These response messages will be handled when the external transition function of the model is executed.

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 2001]. 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 model(s) 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 those new ports. As such, 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, we do find addition of ports having a port type already defined in the current type space useful, since it is likely to have an already defined port with the same type in one of the existing models and it is reasonable for a model to add a port to establish a new coupling with an existing model. Addition of a port with a known type is not included in our operation list above since we plan to add this property in our future work.

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 1. When the simulation starts execution, the models representing the planes send their properties 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 throught the simulation period. 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 2. Since the first two planes send only their updated parameters, which are location and direction, newly added plane will not be aware of the remaining two planes' properties. Therefore a state synchronization is required.

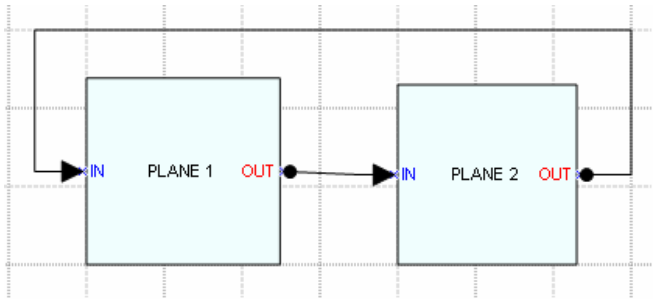

Figure 1 Initial Model

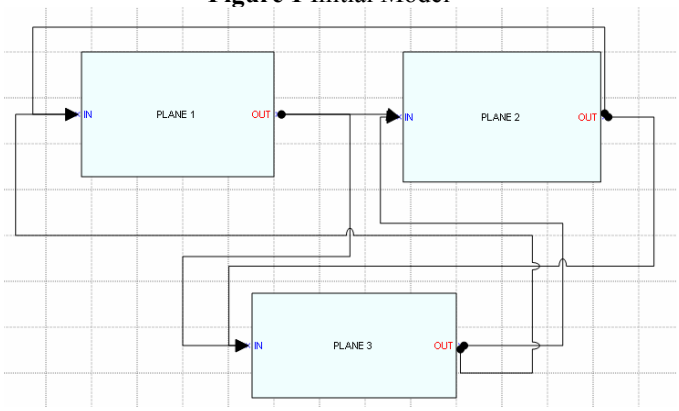

Figure 2 Updated Model

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 an additional implementation overhead on the model developer. A more detailed discussion of the state query mechanism is provided in Section 3.3.

3.2. 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. SiMA run-time layer is implemented in C# programming language but it can interface to models implemented in both C++ and C# programming languages. In this section, extensions to abstract simulators required for executing variable structure SiMA models are described in pseudo code format:

Root Coordinator

```
StructureChangeRequested : boolean
CurrentTime : double

While(simulation end condition not satisfied) Do
    If a model change is requested then
        Process change request
        Send sub-requests to related child
       coordinators
    End If

    <CurrentTime, StructureChangeRequested> ←
       MainModel.GetNextTime()

    Advance time to CurrentTime

    If StructureChangeRequested is true Then
        execute a structural change step
        do state synchronization
    Else
        execute a normal simulation cycle
    endIf
endWhile
```

Algorithm 1 Root Simulation Cycle

After a structural change operation, all models that have new couplings will execute state synchronization mechanism, discussed in Section 3.3, to update their state information.

'ChangeStructure' and *'GetNextTime'* functions of both simulators and coordinators correspond to the case when a message of type *sc* and @ are received from the parent models in [Barros 1998; Himmeelspach and Uhrmacher 2004; Shang and Wainer 2007] respectively.

Coordinator

In *'GetNextTime'* function, next simulation time calculated 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.

```
Function GetNextTime(): <double, boolean>
    tempTime, minTime : double;
    tempSC, structureChangeRequested : boolean;
    minTime ← Positive Infinity

    For each model of the containedModels Do
        <tempTime, tempSC> ← model.GetNextTime()
        If (tempTime < minTime) Then
                minTime ← tempTime
                structureChangeRequested ← tempSC
        Else If (time = minTime and tempSC is true)
        Then
                structureChangeRequested ← true
        endIf
    endFor

    return <minTime, structureChangeRequested>
endFunction
```

Algorithm 2 Recursive Next Time Calculation

```
Function ChangeStructure()
    changeReq : set<operations>;
    For each model of the containedModels Do
        If model requested structure change then
            call model's ChangeStructure function
            add model's change requests to
           changeReq set
        endIf
    endFor

    Process and apply changeReq set
    Send upper-level operations to parent model
endFunction
```

Algorithm 3 Coordinator Change Structure

Simulator

Next transition time and an indication of whether any structural change request exists are sent to the parent coordinator.

```
Function GetNextTime(): <double, boolean>

    return <t_N, StructureChangeRequired>

endFunction
```

Algorithm 4 Simulator Next Time

Structural change function of an atomic model is executed if and only if its next time is imminent and a structural change request has been made by that model.

```
Function ChangeStructure()
    If(StructureChangeRequired is true and t = t_N)
    Then
        call ρ_α
        StructureChangeRequired ← false
        send upper-level operations to parent model
    endIf
endFunction
```

Algorithm 5 Simulator Change Structure Transition

If the state of an atomic model satisfies certain conditions that require structural changes, atomic model marks itself and informs its simulator to initiate a structural change process and this simulator recursively sends this 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. Modifications required for supporting variable structure models can be summarized as follows:

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

3.3. State Synchronization Queries

In SiMA, there is a state query mechanism between connected ports. A port can create a query and send this query to other source ports to which it is connected. This mechanism works in the opposite direction of the normal message flow and it is instrumental in supporting the implementation of variable structure models. It 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 query to gather this information. Implementation details of this mechanism are discussed below.

An interface named '`IStateQuery`' is defined in SiMA as below:

```
interface IStateQuery
{
    Message[] getState(port name);
}
```

This interface has only one member function which takes a port name as the parameter and returns the port's related data. Each model contains a list of '`IStateQuery`' instances for the couplings added in the last structural change step. After all the couplings are added, the states of the models are updated accordingly as illustrated below:

```
IStateQuery[] newCouplings;
foreach coupling in newCouplings
{
    coupling.DestinationPort.QueryState(
        source model, source port)
}
```

Destination ports create queries and send these queries to source ports that are connected to them. When an atomic model receives a query, it sends its state as a response. When a coupled model receives a query, it redirects this query to the source ports that are connected to this port and collects and returns the responses received from those redirected ports. For example in Figure 3 we add a model named 'D' and a coupling from C's 'Out1' port to D's 'In1' port dynamically. After the coupling is added, 'In1' port of model 'D' sends a query to 'Out1' port of model 'C'. Then, 'Out1' port of model 'C' redirects this query to 'Out1' ports of model 'A' and model B. 'Out1' port of model C collects response messages from 'Out1' ports of model A and B and sends these messages back to 'In1' port of model D.

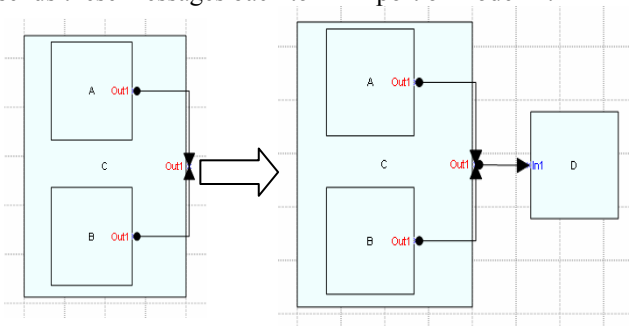

Figure 3 A Model and A Coupling Added Dynamically

When the model initiating the query receives response messages, it adds these messages to the message bag of the port to be handled in the next external transition phase as the following code snippet illustrates:

```
Message[] messages ← sourceM.getState(sourcePort)
messageBag.Add(messages)
```

4. CONCLUSIONS AND FURTHER WORK

We have introduced our approach to implementing 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.

Our approach to add dynamism to our basic DEVS model is similar to that of dynDEVS as indicated earlier. In particular we conform to both dynDEVS and dynNDEVS as the underlying formal specification, with some non-disruptive extensions to the original formal semantics. One particular contribution we offer is the systematic framework support for post-structural-change state synchronization among models with related couplings. Note that we also

benefit from the strongly-typed execution environment SiMA provides.

Although SiMA has been used in a number of simulation applications successfully, our dynamism extensions to our framework are not mature yet. Our objective is to support real world applications in the near future, anticipating that there will be room for improvement to the mechanisms we devised. In particular, scenarios where supporting dynamic fidelity-level adjustments of multiple models in a coordinated way is a requirement, are potential use cases where we hope SiMA would provide a viable solution for application developers.

References

Barros, F. J. 1995. Dynamic Structure Discrete Event System Specification: A New Formalism for Dynamic Structure Modeling and Simulation. In *Proceedings of the 1995 Winter Simulation Conference*, 781-785.

Barros, F. J. 1996. Dynamic Structure Discrete Event System Specification: Formalism, Abstract Simulators and Applications. *Transactions of the Society forComputer Simulation* 13(1): 35-46.

Barros, F. J. 1997. Modeling Formalisms for Dynamic Structure Systems. *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, No. 4, 501-515.

Barros, F. J. 1998. Abstract Simulators for the DSDE Formalism. *Proceedings of the 1998 Winter Simulation Conference.* pp.407-412. Washington DC, USA.

Fujimoto, R.M., Weatherley, R.M. 1996. Time Management in the DoD High Level Architecture. *In Proceedings of the PADS'96,* pp60-67.

Himmeelspach J., and Uhrmacher, A. M. 2004. Processing dynamic PDEVS models. *Proceedings of the IEEE Computer Society's 12th Annual InternationalSymposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04).* Volenlam, Netherlands.

Hu, X.L., B.P. Zeigler, S. Mittal. "Variable Structure in DEVS Component-Based Modeling and Simulation". *Simulation*, Vol. 81, Issue 2, 91-102. 2005.

Kara, A., Bozağaç, D., and Alpdemir, M.N., "Simülasyon Modelleme Altyapısı (SiMA): DEVS Tabanlı Hiyerarşik ve Modüler bir Modelleme ve Koşum Altyapısı", *İkinci Ulusal Savunma Uygulamaları Modeleme ve Simulasyon Konferansı (USMOS),* April 2007.

Ören, T. I. 1991. Dynamic Templates and Semantic Rules for Simulation Advisors and Certifiers. In *Knowledge-Based Simulation: Methodology and Application*, ed. P. A. Fishwick and R. B. Modjeski (eds.), 53-76. New York: Springer Verlag.

Shang, H., G. Wainer. "Flexible Dynamic Structure DEVS Algorithm towards Real-Time Systems". *Proc. Of Summer Computer Simulation Conf.*.San Diego. CA. 2007.

Uhrmacher, A. M. 2001. "Dynamic Structures in Modeling and Simulation: A Reflective Approach". ACM Transactions on Modeling and Computer Simulation. Vol. 11, No. 2, 206-232.

Zeigler, B. P. 1976. *Theory of Modelling and Simulation*. New York: Wiley.

Zeigler, B. P., T. G. Kim, and Lee, C. 1991. Variable structure modelling methodology: An adaptive computer architecture example. *Trans. Soc. Comput. Simul. 7*, 4 (Dec. 1990), 291–318.

Zeigler, B. P., T. G. Kim, and H. Praehofer. 1998. *Theory of Modeling and Simulation*. 2 ed., New York, NY: Academic Press.

Author Biographies

Fatih DENIZ is a Researcher at TUBITAK UEKAE ILTAREN. He received his BSc degree from Bilkent University, Ankara, Turkey in 2007. He is currently a MSc student in Department of Computer Engineering of Middle East Technical University. His current research interests include model-driven engineering and variable structure models.

Ahmet KARA is a Senior Researcher at TUBITAK UEKAE ILTAREN. He has been involved in 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 Department of Computer Engineering of Middle East Technical University.

M. Nedim ALPDEMİR, received his MSc (1996) in Advanced Computer Science and PhD (2000) in Component-Based Simulation Environments from the Department 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 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.

Halit OGUZTUZUN is an associate professor in the Department of Computer Engineering at the Middle East Technical University (METU), Ankara, Turkey. He obtained his BSc and MSc degrees from 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 engineering.