AdSiF: Agent Driven Simulation Framework

Mehmet F. HOCAOGLU

TÜBİTAK Marmara Research Center

Gebze, Türkiye

mfatih.hocaoglu@bte.tubitak.gov.tr

Abstract

This paper summarized a simulation framework called Agent driven Simulation Framework (AdSiF). The motivation of the study is to combine simulation kernel engines with agent technology under a layered architecture to obtain more powerful and flexible simulation products. Bringing agent technologies and simulation technologies together arises many benefits, further enhancing each other by their challenging capabilities, including synergy of being together. Agent technology strengthens simulation technology especially in decision-oriented processes. The properties of Acting proactively, behaving reactively against any event around, living in an environment and affecting it all come from the agent side in an agent-simulation cooperation. Simulation technology supports the cooperation by providing an environment, managing behaviors, managing messages between components and regulating logical time. The framework is more than a simulation framework, it also provides a computational environment and as well as many services for developing applications in a framework based environment using its specific programming paradigm called “state oriented programming paradigm”.

Keywords: AdSiF, Agent, Simulation Framework, State-Oriented Programming

1. Introduction

Cooperation of simulation, artificial intelligence and software engineering technologies present new products, ideas and powerful synergies arising to be together. This study is about one of these products: a simulation framework combining a novel simulation engine solution and agents as decision makers and planners.

While simulation technologies provide behavioral descriptions and related implementations and an environment for the entities, as well as advanced graphic capabilities, agent technologies ensure autonomy, proactivity and opportunity. Autonomy and proactivity, which help to achieve a given task by sensing and reacting events happening in the environment are the main capabilities ensured by agent technologies to simulation entities. Agent technology is strictly related with software engineering rather than artificial intelligence but the flexibility it has makes equipping with decision-making ability easy. Although, being intelligent is not a mandatory requirement for an agent, the ability strengthens the autonomy, proactivity and reactivity that it already has by knowing what to do in any situation. The questions are how to bring simulation and agent technology together and what kind of task partitioning should be applied. This paper gives answers to these questions by delivering decision oriented and knowledge intensive tasks to agents and management tasks of behavioral dynamics to simulation. It also combines all implementations both decision process and knowledge management of agents and behavior, time and event managements of entities under a common language.

The language prepares a standard dialog media for both agents and simulation entities and even for any type of application. The language has its own paradigm called “state oriented programming paradigm”. According to the paradigmatic glance, a real world entity is described by objects as in object oriented programming paradigm and its behavioral description is made by a set of state collections in Finite State Automation, each state of which encapsulates a function that belongs to the entity and may have temporal relations with other behavioral descriptions.

Agent Driven Simulation Framework (AdSiF) is developed based on this paradigm. It has a simulation engine, agents and a series of optional simulation management components such as central memories and event managers. The simulation engine is in charge of time management, event handling and implementations of behaviors including temporally related behaviors. The agents are in charge of decision-making, knowledge generation and management . The event manager is an example of the optional simulation management components and is responsible for managing the actions scheduled such as injecting or purging components into simulation scenarios or from simulation scenarios in run-time. The central memory is responsible to keep and manage decision-making algorithms. User defined predicates are generated by simulation models themselves and/or by the simulation environment. The framework ensures a flexible environment to develop simulation models or any other application. As it keeps the semantic of simulation model behavioral dynamics away from the model, it strengthens model verification actions and also allows modelers to expand their models by external plug in components developed as additional functions or as a whole simulation model.

2. Motivation

One of the main motivations of the study is to develop a simulation framework, strengthening flexibility and reusability by separating simulation behavioral semantic and scenario flow semantic from each other and these promote simulation model verification.

In the literature a software framework is defined as follows:

“A software framework is a partially complete, or semi-finished software (sub-)system that is intended to be instantiated. It defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made. In an object-oriented environment, a framework consists of abstract and concrete classes. Instantiation of such a framework consists of composing and sub-classing the existing classes [Buschmann1996].

In short, a software framework is a set of cooperating classes that make up a reusable design for a specific class of software [Johnson1988; Deutsch1989]. A framework consists of frozen spots and hot spots [Pree1994]. On the one hand, *frozen spots* define the overall architecture of a software system, that is to say its basic components and the relationships between them. These remain unchanged (frozen) in any instantiation of the application framework. On the other hand, *hot spots* represent those parts of the software framework that are specific to individual software systems. Hot spots are designed to be generic. In other words, they can be adapted to the needs of the application under development. When developing a concrete software system with a software framework, the hot spots are specialized according to the specific needs and requirements of the system. Software frameworks rely on the Hollywood Principle: "Don’t call us, we’ll call you." [Larman2002]. This means that the user-defined classes (for example, new subclasses), receive messages from the predefined framework classes. These are usually handled by implementing super-class abstract methods”.

The result of study is a framework called AdSiF. AdSiF has its own architecture and ontological view as well as programming paradigm.

3. Ontological View and State-Oriented Programming

AdSiF presents an ontological view for real world classification. Ontological view is related to describe the real world entities and their relations with environment and other entities. The fundamental description of an entity is that it is an existence that represents behaviors, affects environment and has characteristics making it distinguished. An entity is distinguished from other entities by its attributes, behavioral descriptions, states and messages publishing in the environment. At any time, the entity is in at least a state or more than a state. The state is described not only by values of attribute collection but also an action assigned to the entity to implement and a meaningful naming defining in what condition the entity is. Communication between entities is achieved by messaging and there is no direct method invocation between entities. A message is used to trig a behavior which the sender (originator) wants the receiver to do. An entity executes a behavior by achieving a series of state transitions, each of them encapsulating an atomic action of the entity. The state transitions, namely behaviors represent a meaningful complete activity. Moreover, entities have memories keeping decision making algorithms and knowledge structures (predicates) that decision making algorithms need to give a decision on a time labeled trajectory.

State-oriented programming combines state definitions and relations between states. The fundamental elements of state-oriented programming are classes, states, behavior patterns, behavior pattern containers and messages flowing between simulation models or applications and even in a single component/application body. State-oriented programming has a structure encapsulating object oriented programming. Encapsulation is succeeded by associating methods of classes to states. Semantic structure declared between methods of a class by writing code fragments in Object Oriented Programming (OOP) is described by state diagrams, more specifically Finite-State automations in State Oriented Programming (SOP).

Basically, programming by states is achieved by mapping class methods into states and constituting semantic knowledge that represents a complete behavioral description between states drawing state diagrams. Implementation of an application developed in SOP is to ensure consecutive state transitions. Each state invoked implements the method, which belongs to the application class associated with the state. In SOP, an application is always at least in one state or more than in one state because of other behaviors being executed in parallel.

An AdSiF model has a main class with or without peripheral classes, a database of model attributes and a set of state diagrams (Figure 1). The each set of state diagrams represents a complete course of actions. The state diagrams are similar to hierarchical state machines, based on Harel State charts. It is possible to make a model behave differently by changing its behavior pattern list (BPList). For example, let us assume that the model in Figure 1 is a person model. BPList A is designed so that it consist of what a police officer does such as walking, running, firing, driving, sleeping and may be eating donuts. BPList B is designed so that what a fireman does such as opening a valve, using fire distinguisher, driving fire truck etc. This is the polymorphism in State Oriented Programming.

AdSiF models can be extended by adding new methods in a plug-in format. This is supported even in run-time. After an implementation is started, it is not late to add new methods and attributes to the models being executed in the implementation.

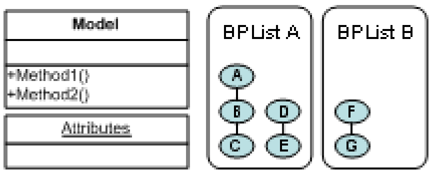

Figure 1: Components of a Simulation Model/Application Developed in AdSiF

In the subsections of this paper, elements of SOP and AdSiF as an implementation of SOP framework are examined in details.

3.1. States

A state is a definite condition, which a component is currently in, such as sleeping, shaking hands, firing etc. Models perform an action by passing into a state, which has an atomic action depending on the detail level modelers expect to see . States are separated from each other by their unique names. More precisely, a state is defined by three groups of descriptors namely operational descriptors, stative descriptors and descriptors of temporal relations.

Operational descriptors, which are guard constraints, methods, messages and duration computers (DCs), are related implementing actions or a computational procedure. The actions may be an operation of a simulation model or may be a computation procedure belonging to an application implemented in the related state.

Guard Constraint is a method the simulation model has with a boolean return type or a predicate defined by users and updated during the simulation model/application execution or a variable limit evaluation. During execution, it is decided whether or not the state is assigned to the model according to the return value of that constraint.

Method is the name of the method that performs the related atomic behavior that we expect from the simulation model or from application to perform in the state.

Duration computers (DCs) are in charge of computing duration for the state. Duration of a state is computed in terms of random value calculation from a theoretical distribution or from an experimental distribution or by reading from pre-prepared files.

While a state can have just one Guard constraint, one Method and one duration computer, it can have many event messages attached for sending. All messages are sent as leaving the state but it is also allowed to prepare and send messages in the method the state is associated to.

Passing from a state to any other state is named as state transition. Models change their states either by consuming the state duration, namely internal state transition or by getting an event from the outside, namely external state transition [Zeigler2000].

Stative descriptor keywords of a state are ***Initial*** and ***Final***. A state defined as initial state is the first state the model is desired to be in at beginning of the simulation execution (at time t_0). Beginning of the simulation execution for a simulation model is the time the model is joined the execution. Since simulation models perform multi-behaviors in parallel, they are in many states at any time in the simulation execution because of the behaviors executed in parallel. By the same reason, it is possible to assign initial states at simulation time t_0. Since behavior patterns are constructed by states that are in charge of performing an atomic behavior, they are organized so that they perform a complete behavior.. One way for the decision of whether the behavior is achieved or not is to define a new state in the behavior pattern called final state. If a model enters a final state, this means the behavior that the final state belongs is achieved and the model sends a message to the component that wants it to perform the action informing it the behavior is resulted successfully. The other way is to declare a "succeed" condition. The succeed condition is similar to the guard constraints. Actually, in AdSiF any logical premises is declared as a method with bool return type or as logical premises kept in component memory and updated during execution in prolog predicate form or as an AdSiF variable evaluation to take a course of action. While constraint methods are invoked to get a truth value, predicates and variables are evaluated. The predicate evaluation includes exploring if the component has knowledge or a decision procedure satisfying the predicate. An AdSiF variable has a limit value and a logical operator. Variable evaluation includes a comparison between its value and its limit value using the assigned logical operator. Whenever a succeed constraint assigned to a behavior is succeeded, the behavior sends a succeed message as the final states do.

A state has a three-phased life cycle. These are entrance phase, action phase and exit phase. The life cycle starts by being set after the related guard constraint has been satisfied. A state can be associated with a set of behaviors by temporal relations. This association is placed into both entrance and exit. This optional property allows a state to activate, cancel, postpone or reactivate a behavior or a set of behaviors depending on how they are associated. In entrance phase, all temporally associated behaviors are implemented. Following this, the method is called and by doing so an action is taken. Until either getting an event message from outside (namely from other components) or elapsing whole duration left for the state, the component stays in the state. In the exit phase, all the behaviors submitted to the exit are invoked and messages are sent. Start of another state life follows termination of a state life, until reaching a last state. If the last state has a limited duration, in other words, its duration is different from infinite; termination of the last state is the end of the behavior life cycle. Getting a message from any other component allowing an external transition also terminates behavior life cycle. In Figure 2, an agenda has two BPs. The model state set defined by active state of active BPs. These are State B and State Z. At that time, the model gets a message "e". In the BPList there is a BP matching the case and it directly activates the BP that has states A-B-C-F-G and the behavior in the agenda is cancelled because of external transition.

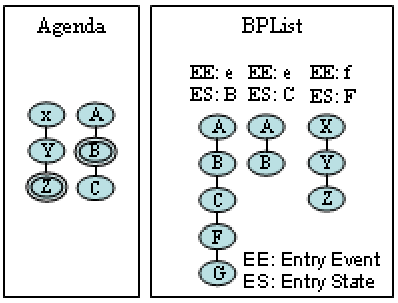

Figure 2: Agenda, BP Selection and External Transition

3.2. Synchronizers

A Synchronizer is a specific type of state. They are differentiated from states in functionality and identifiers they have. Functionally, synchronizers with the same name in different active BPs synchronize the BPs. From the identifiers point of view, they have an empty method to call and a constant type duration computer with infinite time. A synchronizer is placed into a BP as is a state and it prevents to pass to the state following until other synchronizers with the same name of active BPs (in the agenda) are reached. Following all the synchronizers with the same name being set in the agenda, the preceding states in the active BPs are allowed to pass.

3.3. Behavior Patterns (HFSA) and Behavior Pattern List

Behavior patterns are behavior representations similar with finite state automaton. Although, behavior patterns are very similar to the Finite State Automatons (FSAs), specialized state representations are the very essential difference. A BP represents a valid word [Pree1994] and counterpart of a valid word is a complete behavior for the component it belongs. A Behavior Pattern (BP) consists of a series of states and state and event couples called entry event and entry state, respectively.

Activation of a BP performs a complete behavior such as sitting, looking around, walking etc. A BP is activated in three ways.

Event based activation: In this case, the entry event and the entry state works as an activation rule. During execution, if the component gets a message, the BP(s) that accept the event as entry event and one of the model’s active states as entry state are activated.

Conditional activation: A condition is set to the BP being wanted to activate. As soon as the condition satisfied, it is activated. An activation condition can be a method with bool return type, a predicate or an AdSiF variable.

Temporally associated activation: A BP can be associated with a state or any other behavior to be activated.

Implementation of a BP is realized by scheduling it with the simulation model agenda and assigning the states with satisfied guard constraints in consecutive way to the BP, rather than to the component. That means, a BP is always in a state it has. At any time in the simulation, the state assigned is called as BP’s active state and it defines the state of the BP. Since simulation engine manages multiple behaviors in a parallel way, it keeps many BPs in its agenda and the state of the component is defined by collection of active state of the BPs in the agenda. The state set is collection of the active states of the BPs in the agenda. In this sense, if a simulation model, at any time in the simulation execution, has more than one BP being executed, it is not possible to mention about a single state the component is in. The simulation model is in many states at the same time.

It is possible to make BPs temporally associated similar to what it is done in states. A BP is called trigger BP if it temporally activates, deactivates, postpones or reactivates a set of BPs. In this relation, the BP that is managed, trigged by a BP is called associated BP. A trigger BP can trig other BPs in activation and deactivation (cancel) phases. Even a BP can be kept active over the entire course of another BP action. The temporal association is called “During Association” and if the BP associated by during relation has shorter duration than the activating BP, it is repeated until the activating BP is canceled. When the trigger BP is canceled, without taking into consideration in what state the associated BP is canceled. It is possible to give a temporal latency to associated BPs using duration computer. This allows activating BP to activate, cancel, postpone and reactivate associated behaviors in some time to be later computed by duration computer.

Life cycle of a BP starts by activation. Starting a life cycle means being scheduled with the component agenda. External transition or leaving an applicable last state by internal transition ends the BP life cycle. Applicability of a state is explained by having a satisfied guard constraint. AdSiF gives users to keep external transition on or off throughout the simulation execution. AdSiF gives an opportunity to turn external transition "off". In that case, external transition in execution is not allowed. In the "external transition off" case, according to the event received, if the component is in the suitable state (satisfying entry event and entry state match) and it has a BP in its BPList covering the case, the BP is selected but the active BP leading this selection is not canceled and it keeps being active and cancellation of the BP happens because of internal transitions. AdSiF also gives an opportunity to allow making external transition for the selected BPs in "external transition off" case and vice versa. As seen so far, the time step in the simulation execution is determined by BPs. Setting a state determines also the time that simulation model asks for passing. Since AdSiF successes a state oriented simulation language, all task including database management, trace management and some other similar tasks are also represented and executed by BPs. In a simulation scenario execution, BPs which do not have any real world counterparts can not determine simulation time step alone. They always need at least a real world behavior in the model agenda. This kind of behaviors is designated as "dependent behaviors". For example, writing trace reports to a database is not a real world behavior like walking, shooting etc. In this case, the behavior writing is designated as dependent behavior. At any time in the simulation, if it is the only behavior in the agenda or all other behaviors are dependent as it is, simulation does not advance.

3.3.1.Conditions and Properties of a BP

Conditions:

- Activation Conditions
- Cancel Conditions
- Finish Condition
- Looping Condition

A Behavior Pattern List (BPL) is a container that keeps BPs that belongs to the related simulation model. It is suggested keeping the similar BPs in the same BPL so that the BPL represents a complete behavioral capability.

3.4. Inheritance Mechanism

Simulation models have an abstraction level determined in their conceptual model depending on the usage purpose of the model. In problem analysis phase, purposes expected from a model are determined and different model abstraction levels are defined regarding each purpose. As a result, simulation models are developed at different abstraction levels for different purposes.

Inheriting a simulation model from another component ensures to have methods inherited from base model as a benefit of OOP. But this is not enough if the modeler needs to have BP Lists of the base model that is more abstracted and has basic talents than derived model. In this case, the solution is to equip the derived model with the BP list needed. Inheriting BP list of a component also includes having the component attributes if needed. The mechanism is not limited to inheriting BPs belonging to base model, it also allow modelers to inherit any other model BPs and attributes. Inheriting attributes is achieved together with BP inheritance because attributes of models are kept apart from model class in a database. As seen in Figure 3, Model A is base model of Model B and Model B inherits methods belonging to Model A by OOP inheritance mechanism and inherits BPs and attributes of Model A by AdSiF inheritance mechanism. Although there is not any "extend relation" between

Model C and Model B model classes, if it is semantically sound (means if Model B has method with the same name to be invoked by states being inherited), Model B can inherit BPs of Model C as well as its attributes. AdSiF inheritance mechanism makes possible to inherit just behaviors without inheriting attributes. In Figure 3, the BP with states A-B-C and the BP with states F-G are inherited from Model A and Model C, respectively, and its attribute set consists of Model A and Model C attributes (assumed the attributes are also inherited). The BP with states D-E is Model B's own behavior pattern.

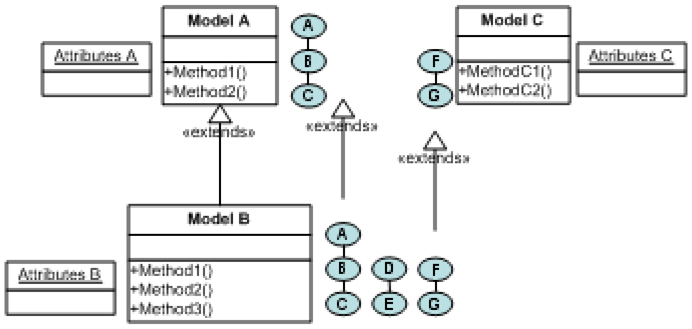

Figure 3: Inheritance Mechanism in AdSiF

3.5. Polymorphism

As mentioned earlier, in SOP, polymorphism is defined as making a model behave in a different way by replacing its behavior patterns list with any other BPList. Polymorphism is also achieved by assigning a BPList to the model or changing it with any other BPList in run-time.

3.6. Variables and Variable Scopes

AdSiF allows users to define their variables during execution. An AdSiF variable is represented as <*name, value, [logical operator], [control value]*>. Logical operator is to evaluate the variable by comparing its value using the operator with the control value. AdSiF has three scopes for variables. These are state scope, BP scope and component scope. A variable defined in state scope can be reached from only that state. A variable in the BP scope is reachable from all the states in that BP. It is reachable from the states of any active BP if it is a component scoped variable.

4. AdSiF Architecture

AdSiF has a layered architecture (Figure 4). Each layer of the architecture serves each other depending on what role it undertakes. The benefits of having a layered architecture especially in simulation environment are discussed by [Sarjoughian2001].

AdSiF architecture is shaped by a simulation engine that is in charge of managing both all simulation models and application behaviors at the kernel and a series of simulation management components including a mission engine managing user applications and user simulation model scenarios, user applications and/or simulation models constructed on the base. Having a real world counterpart is the main difference between a simulation model and an application. In the framework, as is accepted commonly, simulation models are defined as models imitating their real world counterparts and user applications are defined as computational models such as an optimization algorithm, a database application or any other software application.

From the interaction way with architecture kernel and execution points of view, a simulation management component is not different from a user-defined application/model except the responsibility undertaken. Simulation management components which are Simulation Execution Controller, Event Manager, Central Memory, Object Prober, Collision Detector and Mission Engine (Artificial Intelligence Engine), control implementations such as starting and stopping the execution, scheduling events, inserting simulation models/application or interacting with simulation models/applications in run-time, keeping, updating and buffering user defined data structures belonging simulation models/applications in common central memories, counting objects or events flowing in the system and managing collisions in the simulation. Designing simulation management components abstracted from simulation engine as a standard simulation model and being able to manage their behaviors by behavior patterns furthermore providing usage flexibility, it also supports guiding simulation execution itself and defining more specific task definitions controlling whole execution.

As a necessity of being layered [Sarjoughian2001], all components interact with each other via message passing and perform actions by following behavior patterns. In this layered and compartmented architecture, each component undertakes a different mission and serves each other regarding their existence purposes and the specific roles.

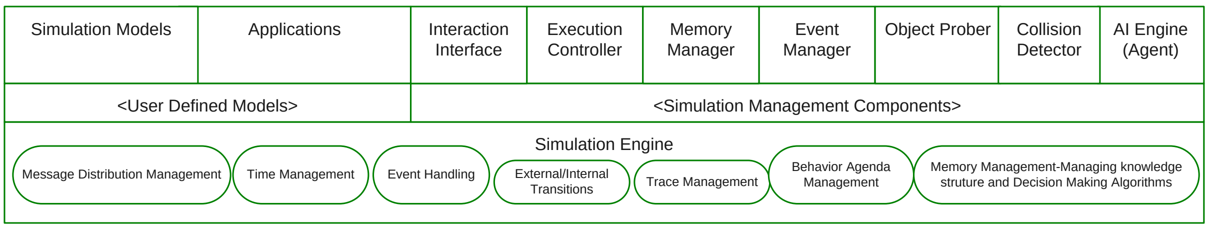

Figure 4: AdSiF Architecture

5. Simulation Engine

Simulation engine is in charge of proceeding with setting and leaving state procedures, managing simulation model internal and external transitions, message distribution, trace management as well as time and event management. Setting or entering a state means that a simulation model changes its state, in other words, it passes from one state to the other. Setting a state to a component or carrying a component from one state to another, consists of a set of "To do" list to be achieved by the simulation engine. First of all, since the decision of setting a state to the model, or from the model point of view, entering a state, is made according to the result of guard constraint of the state, simulation engine invokes the state's guard constraint to make the decision. If the guard constraint is achieved, the state is set to the model and it is designated as a member of current state set. The current state set is defined by the current state of the BPs in the agenda at that time. If the guard constraint is not achieved, the next state is in the BP tried. As soon as a state is set, its method is invoked and handling temporally associated BPs follows method invocation. In standard state representation, IN and OUT represent entering and leaving the state cases, respectively. IN::MUST::Bplist means when the state is set to the component, in other words, the component enters to the state, the actions (BPs) in the Bplist must be activated in parallel. A postponed behavior is frozen in its active state for infinite time. All associated action implementation rules are also valid for leaving a state case. Messages can be attached to a state either temporary or permanent and during leaving the state, all of them are sent. A temporary message is purged from the state after being sent. But as contrary, a permanent message is kept in the state and it is sent all the time during leaving the state if the state is activated.

In internal transition, simulation model passes to the next state from its current state in the active BP because of expiring the duration of the current state. The simulation engine switches the model to next state and manages the "to do list" for leaving state and setting state cases.

In external transition, an event occurrence causes a state transition. Simulation engine checks messages and gets them one by one in urgency order given to them. After removing a message from its message queue, it starts a search process in BPList for the message removed from the queue to select a BP matching the message and one of current state of the behavior in the agenda with its entry state. If it finds a BP in BPList matching its entry state with a member of current state set and matching entry event with the event of the message, the BPs found are activated. Activation means it is attached to the agenda and set its first state to the component by following the state setting procedure.

A message consists of fields for parameters to convey information to the destination model, a destination address (model ID), origin ID (sender address), message type, message ID and, message event to trig behavior patterns. If a message does not have a direction, the message distribution service is activated and by taking user defined message distribution rules, suitable models to get the message are determined and the message is sent to them. The rules are defined according to the attribute values of the models and to the user defined knowledge (predicates).

AdSiF gives opportunities to the user to reach simulation engine kernel. There are simply two techniques. The first technique is using method interfaces such as scheduling a BP directly to agenda, getting simulation logical time, canceling or removing a BP from agenda, attaching a BP created in run time to BP List, freezing, postponing a behavior, changing a behavior properties. It is possible to extend this list longer. These are some of opportunities provided by AdSiF. The second technique is to inherit BPList from the component called Kernel. Inheriting Behavior Pattern List belonging to the kernel gives a key BP set to achieve the tasks enumerated above. If the user does not want to inherit them, simply, he/she can design his/her own BPs using interface methods as state methods and depending on the requirement adding extra functionalities.

6. Coordinator

The coordinator coordinates the execution deciding what long simulation step must be given to simulation model(s). To do that, it asks all simulation models for the time they want to pass, selects the minimum time requirement and gives an advance to the simulation models, which require the same time and calculate the new simulation logical time. In this sense, real simulation logical time is known by just the coordinator. All other simulation models know their last advanced time. Models requiring a longer time to pass wait for time advance until their requirement getting sooner. By that time they repeat their requirement at all steps.

7. Sub-Simulation Clusters

A simulation model can have sub-models. A sub-model is a model attached to a model and managed by it. Sub-models do not have any direct interaction with the coordinator. They get time advance from the model they are attached to. This allows modelers to create new models in run-time and managing them directly.

8. Simulation Management Components

Since AdSiF has a component based and layered architecture, some tasks regarding simulation execution such as inserting a new model to the simulation scenario, sending messages to models, interacting with models and managing and leading scenarios of models etc. are performed by the models called Simulation management components. Because of their abstracted nature, it is possible to equip with new capabilities and even replacing them with newer models, because AdSiF provides interfaces allowing modelers to reach even simulation engine kernel. Any type of simulation components are accommodated in a scenario in any number without any limitation.

8.1. Mission Engine

Mission engine is a Belief, Desire and Intension agent [Wooldridge1997; Wooldridge2000; Russel1995]. It is developed based on Firby's Reactive Action Package (RAP) system [Firby2000] by translating it into state oriented environment and equipped with more developed query and decision making capabilities. The first step in preparing a mission for the model is to create a Reactive Action Package. The main difference is to attach messages to the primitive action placed at leaf of RAP. The messages are sent to the model to invoke an action represented as a BP. Another important difference is in queries applied to the packages. The queries are designed as a model class method, a predicate or an AdSiF variable. In execution, RAPs are translated into BPs and are executed by the simulation engine. A main mission engine creates a sub-mission engine for each simulation model in the scenario that has a mission to implement and attaches itself as a subcomponent. Every sub mission engine implements a different mission for the model it works. This is achieved by polymorphism and sub-simulation cluster properties of AdSiF. After expanding a RAP, the last step is to send a message to the related model to make a task done, and wait until getting a succeed message from the model. Translating RAPs into BPs ensures all State Oriented AdSiF advantages in use including parallel mission execution.

8.2. Event Manager

The component handles scheduled events through the simulation execution. Scheduling means planning a series of action occurrences following a time plan or at certain times with occurrence limits or an occurrence number.

Scheduled events are categorized under four groups.

Inserting a component into simulation: The planning starts with declaring what component to be inserted to simulation in run-time. The plan consists of what component to be inserted, in what time periods and in what numbers

Removing a component from execution: it consists of when the model will be removed.

Sending messages to components: it consists of when the message will be sent to what model.

Setting actions to keyboard and mouse: it consists of what message to be sent to the model by mouse or keyboard indication.

Event Manager translates the schedules into the BPs and activates them. Simulation engine executes the BPs.

8.3. Memory Manager

The component keeps the global user defined data structures. The user defined data structures are a collection of selected data generated by simulation model during execution formatted in predicate form. For example, the data structure *location(modelID, latitude, altitude, longitude, logical Time)* has a name like location and a parameter set like written in parenthesis. In this example first four parameters are models attributes and their values are got acquiring attributes value in run time and for the last parameter a method giving logical time are invoked and its value is set by return value. At any time in execution, the predicate *location(platform1, 100, 120, 300, 220.3)* is sent to the memory manager component.

Memory manager also stores and allows modelers to use decision making algorithms developed in Prolog. In run time, it is possible to store new decision making algorithms.

8.4. Simulation Execution Controller

Simulation execution controller controls simulation execution stop, start and pause operation. Since it is possible to define new methods (as plug-ins) or to design a new execution controller from scratch and to design new BPLists, the component gives us an opportunity to implement actions such opening or closing databases, writing some information for specific period of the execution such as stop, start and pause.

8.5. Trainer Console

Trainer Console allows simulation scenario designers and scenario execution controllers to interact simulation models in the scenario being controlled. Simulation models, which are allowed to be controlled by switching related attribute of the model on register themselves to the trainer console that are addressed. Any simulation model can register itself to any trainer console in the simulation scenario. Registration consists of sending registration message carrying BP names, entry event name of the model. The users can send messages to the simulation models using this information and gain an opportunity to make the models do whatever they want. Moreover, they can manage the behaviors simulation engine provides as a kernel function such as attaching a subcomponent, making simulation model message box blocked to prevent it to get and process the messages, freezing the model ensuring postponing all action in its agenda etc.

8.6. AdSiF Debugger

AdSiF debugger provides an interface to browse simulation model agenda and allows users to run simulation execution step by step. Since model agendas are showed in different windows, this ensures users to check consistency of BPs out.

9. Discussions

AdSiF is a computational environment and further a simulation framework. The framework ensures a tremendous flexibility, because it allows expanding models by adding new methods as plug-Ins and adding/removing attributes keeping model attributes out of the model. Providing models to interact each other by messaging makes models abstracted and hence reusability arises. AdSiF not only supports the design and programming of the OO model, it also provides a graphical programming environment in state oriented programming. Flexibility, reusability and being component based are AdSiF's software engineering perspective. On the other side, from the simulation perspective, AdSiF provides some opportunities. In short, these are handling zero-time-advance, keeping semantic out of the model and support to the conceptual model development. Handling zero-time-advance provides to propagate the non-real world actions to BPs as states. That means designer can make the model process these action without advancing simulation time such as publishing a message, getting a response from users etc.

Inheritance mechanism of AdSiF supports to develop hierarchical modeling. At each hierarchy, an abstracted model is placed with its corresponding BPs. Having models at different abstraction layers with their behavior pattern sets covers different level of simulation requirement at each abstraction layer.

AdSiF covers different user profiles, which are modelers, developers, scenario designers, domain experts and ordinary users in different expertise levels, for designing and executing scenarios.

10. Future Studies: Executing Multi-Scenarios and AdSiF UML Diagrams

In the near future, a new version of AdSiF supporting execution of multiple scenarios in parallel will be developed. In this version, a coordinator of coordinators that coordinate single simulation scenario execution will be developed and the coordinator also will support knowledge sharing between simulation scenarios.

The next step is to develop standard event, state, BP, BPList representations for UML. This will be a state-oriented programming environment.

11. References

[Buschmann1996]-Buschmann, F., "Pattern-oriented software architecture: a system of patterns", Chichester ; New York, Wiley.

[Cassandras1993] – Cassandras, C. G., "Discrete Event Systems: Modeling and Performance Analysis", Aksen Associates Incorporated Publishers, (1993).

[Deutsch1989]-Deutsch, P. L., "Design reuse and frameworks in the Smalltalk-80 system", Software reusability, volume II: applications and experience. T. J. Biggerstaff and A. J. Perlis. Reading, MA, Addison-Wesley: 57-71, (1989).

[Firby2000]-Firby, J. Fitzgerald, W., "The RAP System Language Manual", Version 2.0. Evanston, IL, Neodesic Corporation, (2000).

[Johnson1988]-Johnson, R. E., Foote, B., "Designing reusable classes." Journal of object-oriented programming 1(2): 22-35, (1988).

[Larman2002]-Larman, C., "Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process", Upper Saddle River, NJ, Prentice Hall PTR, (2002).

[Pree1994]-Pree, W, “Meta patterns - a means for capturing the essentials of reusable object-oriented design”, in M. Tokoro and R. Pareschi (eds), Springer-Verlag, proceedings of the ECOOP, Bologna, Italy: 150-162, (1994).

[Russel1995]-Russell, S. J., Norvig, P., “Artificial Intelligence: A Modern Approach”, Prentice-Hall International Inc., New Jersey, (1995).

[Sarjoughian2001]-Sarjoughian, H., B. Zeigler, S. Hall, “A Layered Modeling and Simulation Architecture for Agent-based System Development”, IEEE Proceedings, 89(2): 201-213, (2001).

[Wooldridge1997]-Wooldridge, M., “Agent-based Software Engineering”, IEE Proceedings on Software Engineering, 144(1): 26-37, (1997).

[Wooldridge2000]-Wooldridge, M., “Reasoning about Rational Agents”, The MIT press, (2000).

[Zeigler2000] - Zeigler, B. P, Praehofer, H., Kim, T. G., “Theory of Modeling and Simulation”, Second Edition, Academic Press, Florida, (2000).