International Journal of Advancements in Technology

Hocaoglu, Int J Adv Technol 2017, 8:3
DOI: 10.4172/0976-4860.1000186

Research Article Open Access

Aspect Oriented Programming Perspective in Agents and Simulation

M Fatih Hocaoglu*

Department of Engineering and Natural Sciences, Istanbul Medeniyet University, Istanbul, Turkey

Abstract

While object-oriented programming paradigm gives a vertical software design, aspect orientation enhances this vertically deep design by horizontal association. An agent based solution is offered for aspect-oriented programming paradigm in agent and simulation development. Agent driven simulation framework (AdSiF) is technological background of the study. AdSiF provides a declarative scripting agent programming language and it combines object oriented programming, logic programming, and agent based programming in state oriented programming paradigm as a surrounding paradigm. State-oriented programming paradigm is at the background of script and it allows programming by extended state charts. In this paper, the solution given by AdSiF for aspect-oriented programming paradigm that draws a solution background related with scattered codes, scattered requirements and tangled requirements is examined. It is explained how to distribute states and behaviors to satisfy scattered requirements, to behaviors and behavior lists, respectively and how behavior phase transitions are used to activate specific behaviors and behavior lists (a group of behaviors) to satisfy a set of requirements and to show different behavioral aspects of an agent and/or a simulation entity. The solution also provides a solution by shifting modeling aspects conditionally in run time for conceptually different modeling requirements as well as tangled requirements. From this respect, the solution carries aspect oriented programming from design time to execution time and provides a dynamically manageable, flexible, loosely coupled and high coherent simulation and agent design. Using dynamic aspect management, parallel simulation synchronization algorithms are modeled as behaviors and each of them is grouped as an aspect and a rule based reasoning mechanism is developed to shift between algorithms depending on control criteria.

Keywords: Aspect oriented programming; Declarative language; Dynamic aspect; Script programming; Time Warp

Introduction

AdSiF can be defined in general terms as a declarative scripting language. The structure of AdSiF is a combination of multi programming paradigm and state-oriented programming (Hocaoglu, 2005). The structural integration of Object-oriented programming (OOP), logic programming, Agent-based programming, and Aspect-oriented paradigms define ontological aspect of AdSiF. The paradigms covered by AdSiF as well as reusability, interoperable and flexibility properties are in accordance with the world view. The integration of each paradigm with its world view provides a significant infrastructure for agent-based simulation modeling. This approach ensures that the application will be able to responsive with the characteristic of behavior planning and has autonomous and anthropomorphic ability, at the same time; simulation models will be more social, flexible and interoperable. Due to fact that the models reach available information in their environment and make reasoning based on collected information are two of main advantages of logic programming paradigm. Hence, it provides a Dual-world representation for simulation models.

In this study, the AdSiF solution approach of aspect based programming paradigm is handled. Then, the conceptual model of solution and its contributions to the modeling techniques are discussed and exemplified. Thereinafter, the background information about aspect-oriented programming paradigm is explained comprehensively. In section 3, the solution of aspect-oriented programming paradigm, in section 4, a case study in the field of simulation will be provided. Lastly, the benefits of AdSiF aspect-oriented programming are dealt with in the conclusion part.

Motivation

The motivations of the study are summarized under three categories. These are 1) to give a programming approach based on agent programming to aspect oriented programming, 2) to propose a solution for simulation time management modeled as an aspect and not coded into simulation kernel to make simulation execution algorithms plug-play loosely-coupled software components, 3) to manage simulation time synchronization in parallel/distributed simulation as a behavioral aspects. A time synchronization behavioral aspect consists of a set of behavior that manages time synchronization among simulation entities, time management, and event management.

Definitions

State: A state is defined as a definite mode in which a simulation model/an agent is. States are the most atomic element in agent and simulation model development in AdSiF. A state has a set of phases named as entry phase, exit phase, and external transition phase. A function is assigned to each phase and it is called if the state is in the related phases. A state may have, instead of sub-states, temporally related behaviors. A state sends event attached to it in exit phase and it is possible to make a state connected with any other behaviors based on its phases. This allows a state to activate, cancel, suspend, and/or reactivate other behaviors in the phase they are connected and this is named as temporal relation.

Behavior: A behavior is constituted by a series of states, temporal relations, drive conditions, a guard constraint, trigger event-entry state couples, phase functions, drive conditions, and events attached. A behavior is defied so that it success a specific goal or gives a meaningful result. For example, if a "step" is defined as a state, "walking" is defined as a behavior and it is constituted by the state "step".

Behavior list: A behavior list consists of a set of behavior. A simulation model or an agent may have more than one behavior list but

***Corresponding author:** Dr Mehmet Faith Hocaoğlu, Faculty of Engineering and Natural Sciences, Istanbul Medeniyet University, Istanbul/Turkey, Tel: +90 216 280 33 33; E-mail: mfatih.hocaoglu@medeniyet.edu.tr

Received: April 21, 2017; **Accepted:** May 22, 2017; **Published:** May 29, 2017

Citation: Hocaoglu MF (2017) Aspect Oriented Programming Perspective in Agents and Simulation. Int J Adv Technol 8: 186. doi:10.4172/0976-4860.1000186 at least, they must have one behavior list active. It is possible to execute behaviors that are in an active behavior list.

Aspect Oriented Programming

Aspect-oriented programming (AOP) aims to categorize different design objectives and to structure modular software. The essential interests of aspect-oriented programming are especially scattered requirements and tangled requirements. Scattered requirements can be accepted as a significant obstacle to modularity. In the approach simulation environment, a model is interpreted that it is able to categorize anticipated behaviors by modeler for different conceptual world and resolution [1-3].

Aspect-oriented programing solution is considered as a solution which starts from requirement phase and consists of coding and modeling level includes analysis and architecture design. The AOP does not introduce a completely new design process but just a new means to enhance design [4]. As procedural programming brought functional abstraction and object oriented programming gave birth to object abstraction, aspect-oriented programming introduces concern abstraction [4,5].

Aspect-oriented programming has been seen as an important support in software metrics especially in terms of modularity, simplicity and readability [6]. There has been relatively limited number of studies available in the literature on computational aspect of AOP parameters [7]. Kersten and Murphy have showed in NPY applications the success of AOP with the practical implementations and developed codes of practice [8].

In the literature, Aspect J-Like and Hyper J-Like solutions are commonly preferred. The essence of Aspect J-Like approach is defining point cuts. In the Hyper J-Like approach, the combining of state graphs belongs to independently developed models are provided and it requires refactoring differently from Aspect J-Like. In the article of Walker et al., an initial view of usability metrics for AOP has been provided [9]. Soares et al. have indicated the high performance of Aspect J applications particularly in web-based applications. The most remarkable study among other researches is carried out by Garcia A et al. [10], due to fact that it is directly related to our application area, AdSiF, which is agent-based application and provides agent programming language. They provide a computational assessment between template-based approach and Aspect-based programming for multi-agent programming systems. In this study, it is also concluded that agents provide an advanced modularity for cross cutting requirements.

Among previous studies on the integration of OOP design template-based software development and AOP, Vaira and Čaplinskas concentrate on generating design templates independent of the software paradigm. Additionally, they claim that design templates which can be applied to different paradigms are also available [11]. Tsang et al. have evaluated the interest separation performance of AOP. In the study, CK metrics [12] is utilized and the comparison of OOP and real time systems of AOP is clarified. In the result of this investigation, a developed modularity and a reduction in cohesion are observed [13]. The work on cohesion measurement properly in AOP is undertaken by Kumar et al. [14] and they provide a generic framework to define cohesion.

Not only studies on generating abstracted designs and separation of design aspects created by AOP but also some studies especially pointing to solve semantic confusion between the designs of aspects are carried out recently [15]. Also, it is investigated from article that the coordination and cohesion of design aspects can be improved in parallel with reusability, modularity and extendibility metrics. A considerable amount of literature has been already published on modularity and much more information has become available to modularity strengthen [16].

In the application of safety and mission critical system includes non-functional requirements like error tolerance, tangled and scattered non-functional requirements are provided by AOP design pattern. These patterns are improved for error detection, error management, recovery mechanism and leakage controls in safety systems [17].

The essential approach in use of AOP in AdSiF is accepted as a model integrated computing (MIC) solution platform [18,19]. Generally speaking, in model integrated computing environment, the main concern is model and a software system is seen as a collection of models at different levels of abstraction. Every collection represents a different perspective of the system and from this perspective, each engineering task is considered to be the description of a model [20].

AdSiF Aspect Based Programming Solution

In the AdSiF modeling approach, a model contains all modeling perspectives and each of the different modeling perspective is considered as a behavior category. All the behaviors that an agent and/or a simulation model have are grouped under logically separated categories [21]. A category consists of semantically close behaviors. For example, behaviors related with moving capability for a human model are collected in a specific behavior category by separating them other behaviors such as capabilities of cooking, reading, and writing. Separating semantically different behaviors from each other into categories gives modelers the ability to manage models according to behavioral aspects. Furthermore, AdSiF provides a solution for the scattering requirements by scattering behaviors or states which meet the requirements to behavior and behavior sets. Aspect J that has similar perspective executes point of injections in the time of synchronization status operating, internal state transition, behavior triggering and event sending. This provides a solution with low cohesion and high consistency [22]. The thing done in the solution approach is to separate different behavioral aspects into behavior lists (categories) instead of modeling each as a single model. A behavior list can be defined as a set of behaviors which provides a particular modeling perspective and behavior lists are arranged to meet requirements that they aim to satisfy. A model has at least one active behavior list and the number of active list is not limited by one. A behavior list is activated or deactivated depending on the conditions that are attached to them. Activation of a behavior list containing behaviors that provide a group of requirements depends on the satisfaction of the relevant activation condition. Similarly, deactivating a behavior list means there is no need for the behaviors any longer that the list contains or necessary conditions to execute the behaviors are not satisfied. To satisfy structurally different requirements is achieved by keeping the behavior lists that satisfy the related requirements active.

Aspect separation carries out significant flexibility to manage cohesion and contradiction. Keeping mutually exclusive behaviors from each other (i.e., that causes conflict from time to time) in separated behavior lists, and being able to use the behaviors inherited from root (parent) models in derived models provide high flexibility.

Behavior lists and inheritance

In this section, how a behavior list is extended by any other behavior lists and how scattered and tangled requirements are satisfied are shown.

As shown in Figure 1 FsaListBase is defined as *public* and it is extended by FsaList.0 and FsaList.1 It has two behaviors namely, *Behavior.C* and *Behavior.D* which is defined as *private* and *public,* respectively. A *Public* behavior is created by derived behaviors using inheritance and considered as a behavior which is able to perform functions executed in both two behavior lists (parent and derived). In the Figure 2, there are two separate behavior lists named as FsaList.0 and FsaList.1, which consist of different behavior sets belonging different model aspects. Logical expressions called *Act.1* and *Deact.1* enable *FsaList.0* behavior list and passivates, respectively, at any time they are satisfied in run time. FsaList.0 is set as startup behavior list of the relevant model and its initial behavior is set as *Behavior.5*.

Managing scattered requirement

AdSiF provides a set of solutions to satisfy scattered requirements based on concepts of states, behaviors, and behavior lists. These are:

· State distribution: A state invokes a set of functions that satisfy a scattered requirement in its related phases. It is placed to the behaviors so that the behaviors activate the state to satisfy a requirement such as logging operations.

· Behavior and state distribution: A whole behavior that satisfies a set of specific scattered requirements is put into behavior lists. The behavior is activated anytime that is needed by an activation condition or an event received.

· Temporal relation: The usage given below is based on a behavior that satisfies a scattered requirement set. In his usage, a behavior directly invokes the behavior that satisfies a scattered requirement set, in the phase, the behavior being activated, attached. As seen in Figure 3, *Behavior.0* activates *Behavior.1* in its *State.0* entry phase.

· Behavior conditional phase transition: It is possible to determine phase transition condition including activation condition of a behavior depending any other behavior state transition. In Figure 2, it is seen that a set of phase transition conditions is attached to the behavior *Fsa_logging*. The first and the second conditions mean if any behavior enters a state transition named *State_0* (state enter phase transition) or exits from *State_1* (exit phase), the behavior *Fsa_Logging* is activated. It is assumed the behavior *Fsa_Logging* is a behavior that satisfies a set of scattered (crosscutting concern) requirements. Similarly, as seen from the third condition, if the state *Step,* which belongs to *Fsa_Step*, the behavior *Fsa_Logging* is activated (driveType) after a duration determined by *attribute0,* if and only if the condition given in the guard block (if *Func_D* returns true) is satisfied. The fifth condition means if the behavior *Fsa_Analyze* is finished the behavior *Fsa_Logging* is activated. It is also possible to declare a guard for all declarations as is done in the third declaration. As an opposite approach, in Figure 3, if the behavior *Fsa_A* enters *State_1*, the behavior *Fsa_B* is activated. The usage allows modelers make more specific definitions. The opportunities make aspect programming a declaration issue, not a function programming issue. The main point is that scattered requirements are met by behaviors placed into behavior lists or states placed into behaviors that invoke functions that satisfy the requirements. The behaviors have their own semantics that are being able to be overridden in any simulation entity/agent situated in inheritance hierarchy and also behavior lists have activation and deactivation conditions providing immediate reactions to the requirements arising in run time. Being able to satisfy a scattered requirement by a set of function wrapped by a state or a behavior provides designers a state and behavior phase based function execution. The designer may define a specific state transition phase to execute a set of function and he/she wants it to execute any time the phase transition happened. The conditional phase transition does not intervene model behaviors and functions because it is an indirect invocation.

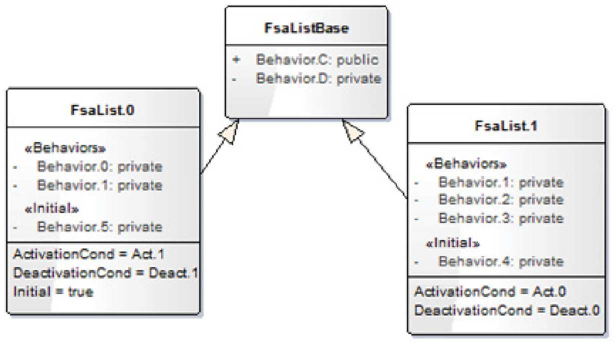
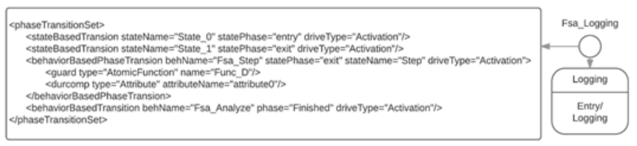


Figure 1: Structure of behavior list.



```
<phaseTransitionSet>
    <stateBasedTransion stateName="State_0" statePhase="entry" driveType="Activation"/>
    <stateBasedTransion stateName="State_1" statePhase="exit" driveType="Activation"/>
    <behaviorBasedPhaseTransion behName="Fsa_Step" statePhase="exit" stateName="Step" driveType="Activation">
        <guard type="AtomicFunction" name="Func_D"/>
        <durcomp type="Attribute" attributeName="attribute0"/>
    </behaviorBasedPhaseTransion>
    <behaviorBasedTransition behName="Fsa_Analyze" phase="Finished" driveType="Activation"/>
</phaseTransitionSet>
```

Figure 2: Indirect phase transition.

To be able to see in more detail, we can examine the example given in Figure 2. As seen in the figure, the function *Func.0* is called by *State.C* and the state is distributed the behavior in behavior lists. Similar way, the behavior *Behavior.1*, which satisfies a specific requirement or requirements, is distributed into behavior lists and it is connected to a specific phase of behavior or states or an activation condition defined for the behavior. This ensures to activate the behavior a set of certain situations defined by modelers.

Behavior.1 and *State.C* in *FsaList.0* are activated by internal state transition, whereas *Behavior.1* that is triggered by an event in *FsaList.1* and *State.C* processes a set of functions. By this design, software design deepening in vertical axis is extended and associated with the horizontal axis (Figure 4).

In addition to the solution, if a model is composed with other models as composition or aggregation, the compound model undertakes time and event management of sub-models. In the design presented in Figure 5, a composition relationship between *Model.A* and *SubModel.0* is defined. *Model.A* is associated *Submodel.1* and *SubModel.2* with aggregation type of relation. *Model.A* is incharge of time management and event handling (distributing event to the sub models, not processing them) of the models it is aggregated. When it is considered that each combined model manages different aspects, as is done in MIC. In this sense, distributing aspects into models is carried out by a higher abstracted model.

Each behavior list is designed to meet a set of requirements. As mentioned earlier, it is possible and offered to design a behavior list with a well-defined purpose aiming to satisfy a bunch of requirements connected each other. Since they may have activation and deactivation conditions, it is possible to react any requirements at the time they arise. This is also named as dynamic aspect management.

Dynamic aspect management

Dynamic aspect management is related with being able to activate

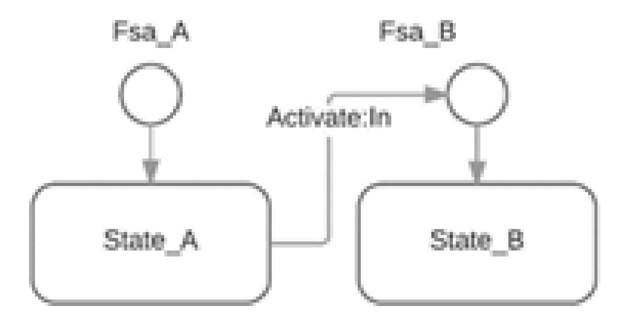


Figure 3: Direct activation with temporal relation.

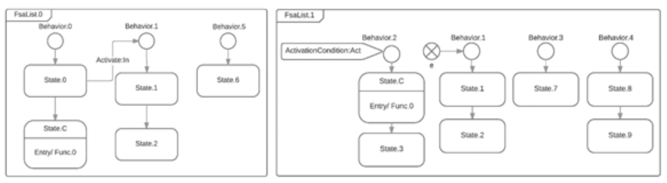


Figure 4: Aspect management with behavior list.

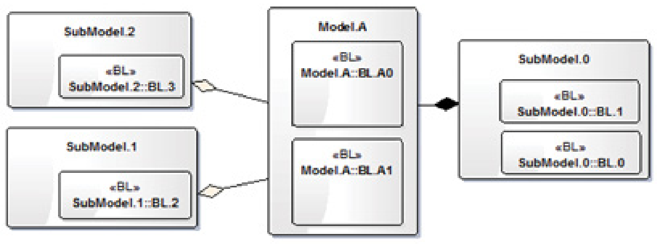


Figure 5: Compound models.

or deactivate behavior categories in run time depending on constraints associated with them as an activation condition and deactivation condition, respectively. Since each behavior list or a set of lists represent an aspect, an activated behavior list transits to the aspect that suitable for the state that is defined by its activation condition. An activated aspect dynamically loads the function libraries that consist of functions to be reached by the behaviors. A deactivation condition also works as similar to an activation condition and it deactivates the aspects that are not suitable for conditions or the states that the entity is in.

Aspect management with relation

A relation is defined between two entities and it has two sides named left side and right side. Both sides may have more than one entity. A relation is defined depending on application domain conceptual model. For example, the sentence "*A commander commands the soldiers*" consists of a relation such as "commanding" between commander and soldiers or "*An F16 carries Type-A missiles*" defines a relation between F16 and type-A missiles. It is possible to make definitions more general based on their inheritance structure. A relation defined over root level entities is valid for the derived ones. A relation serves basically for two tasks.

1. Messaging: The main point in messaging, to provide a mechanism for the entity that sends a message without necessity to know exactly what entity to be sent. Let us assume there is a relation between two entities named *Rel*. The entity sending a message says "*I am sending this message (event) to the entity that has a relation named Rel with me and located on left or right side of the relation*" instead of saying "*I am sending this message (event) to EntityA*". This provides an opportunity to be able to change the entity that receives message without making any change on the entity that sends the message. In other words, the relations establish a loosely coupled interaction between entities.

2. Behavior management: It is possible to declare a set of behavior lists and actions to activate, any time a relation is established between two entities or detached. The declarations are done for both sides and consist of what behavior category or categories are activated and what actions are executed. Establishing a relation may be seen as an activation condition for an aspect and detaching is vice versa.

Especially, behavior management with relation concept is strictly related with dynamic aspect management, since it can be used as an aspect activation and deactivation constraints.

Aspects and Simulation Concerns

The simulation concerns that are separated using aspect oriented programming approach are examined under two categories. The first category is named as simulation execution and it consists of synchronization, event scheduling policies, optimization, and distribution [4]. The second one is about business layer. Business layer means what we expect from simulation entities do. While the simulation layer concerns deal with execution of simulation itself and it is independent from any implementation domain, the business layer is strictly related with implementation domain and the focus is to manage tangled and cross-cutting requirements by arranging behaviors so that entities have behavior lists that consist of behaviors satisfying a group of requirements and behaviors and/or states that satisfy scattered requirements such as logging, security, graphical representations are scattered among different behavior categories.

Simulation execution concerns

The concerns taken into consideration under this category are established by many simulation tools more direct way and it is mostly not possible to change their solution approach. In this study, the main motivation is to make event scheduling policy, synchronization algorithms (parallel simulation execution), and distributed execution changeable, even in run-time. The main idea is to switch from an aspect to another depending on the condition the entity is in. An aspect may consist of a set of behavior either belonging domain (business) or belonging simulation management. Since the aspects that consist of managerial behaviors have higher importance, the behaviors that they have are executed before the domain behaviors. That means the simulation management functions are run before the domain functions. If a modeler desires to change the simulation management algorithm, he/she defines activation and deactivation conditions for the behavior lists that each consist of a specific simulation management algorithm such as optimistic simulation algorithm, conservative algorithms etc. The behavior list with satisfied activation condition is activated and it undertakes the simulation management. The fact that simulation management functions such as collecting and distributing events, requesting interval state transition times, requesting temporal relation time between behaviors are common function and a state surrounding a set of functions are used in different behaviors representing simulation management and they are located in different behavior lists. To be able to design our own time warp algorithm, a set of behaviors, which execute the algorithm, are designed and they are put into a behavior list with an activation condition and deactivation condition. For each management algorithm, a behavior list is designed and they are grouped as an aspect.

Time management and synchronization: Time warp mainly focuses on time management and event handling to be able to satisfy local causality constraints. For parallel/distributed simulation, it guaranties that a parallel/distributed simulation execution gives the same result with a local execution. For both local and parallel/distributed execution, the algorithms for time management and event handling are designed as behaviors. Because of their high importance, the behaviors are processed before the domain behaviors. The management algorithms are collected in separated behavior lists. The management behaviors are interpreted by core script engine as is done for all behaviors. Modelers may design different time management, event handling and synchronization algorithms defining their algorithms as behaviors.

In Figure 6, a time advance based time management algorithm is modeled as a behavioral aspect. Atomic actions for time management such as collecting time requirements (by the function *transitiontime*), collecting and distributing events (*ScheduledEventTime*), handling temporally related behaviors (*TemporalRelationTime*) (activating, cancelling, suspending, or reactivation a behavior from a state or any other behavior), and determining event processing orders are called from management behavior states for a local simulation execution. Until all time requirements are determined and the minimum time selected, simulation time advance is not allowed (*SetAdvanceTime).* This is achieved by the synchronization object *Sync*. The events to be processed are sorted according to their importance criteria and importance parameter is calculated by a function declared by modelers. While the algorithm is based on scheduled events and state transition times (including temporal relations), it is possible to design a set of behaviors as another aspect based on next event time. In this case, next time to advance is determined according to both the next event time stamp to be processed and state transition times. For next event scheduling, optimistic and conservative approaches are applicable for local executions and distributed simulation. For parallel/distributed simulation, time warp implementation is shown here (Figure 7) for roll back mechanism [23] as an implementation of optimistic approach. Rollback mechanism, as seen in Figure 7, is given below:

· Check event to be processed and choice the event with the closest time stamp to the simulation clock (Fsa_TimeAdvance "Check Next Event" state),

· Is the event time is earlier than the simulation clock (Activat e:In:Guard(SimulationClock>EventTimeLabel)?

· If yes, then roll back the simulation to an earlier time point than the event time (Δe) (Fsa_RollBack),

· If no, Check internal transition time (Δint) (Fsa_ StateTransitionTime) and temporal relation time (Δtr) (Fsa_ TemporalRelationTime),

· Find minimum time min={Δint, Δe, Δtr},

· Grant minimum time (State "Next Time"),

· If the current simulation time is equal to the event time, then consume event,

· Save simulation state vector,

· Go to the first step.

It is possible to design a behavioral model of more other time wrap algorithms such as Null message passing, centralized barrier, tree barrier, butterfly barrier [23], etc. It is contended giving an example to show how a time warp algorithm is modeled as state diagram and how it is ensured it manages simulation execution.

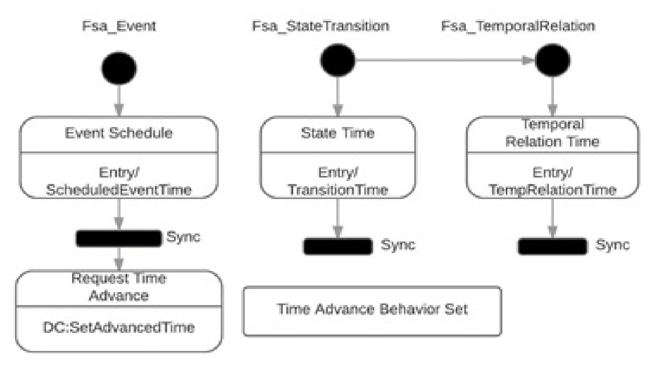


Figure 6: State time advance.

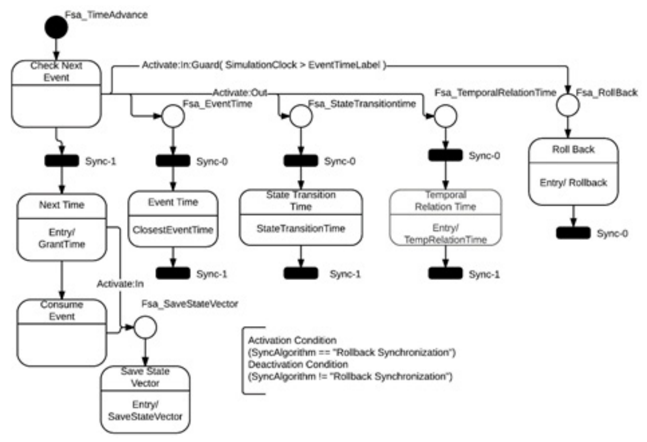


Figure 7: Roll back mechanism.

Business concerns

At business layer, depending on simulation conceptual mission model, simulation entities categorize their behaviors depending on requirements and they activate or deactivate according to the constraints attached to each. Activation and deactivation constraints are determined based on environment conditions and state of entities. Business layer concerns are placed at the top layer of the simulation concerns and the behavior categories and behaviors they consist of are executed after behaviors and behavior lists that of simulation concerns because of their low precedency level comparing with simulation concerns. Examples for business layer aspect definition are given in Section 7.1.

Application Categories for Simulation

Simulation applications are categorized under two main headlines as analysis and training. In analysis-purposed simulation application, modelers rarely need real-time and user-interactive operations. The main expectations are having as fast as possible execution and having logging operations. Access to the highest possible speed, taking simulation logs at desired level of detail, and under the required conditions/time are a significant requirement set for analysis-purposed simulations.

In training-purposed simulation applications, being real-time, graphical user interface, and user-interaction are often encountered as more important requirements. Although, engineering calculations performed basically are the same way, varying requirements necessitate different functions for both categories. Solution in training-purposed simulations, active behavior lists, categorically, consists of interface control utilities, and execution speed control whereas in analysis-purposed simulations, high resolution logging behaviors and the behaviors related control of the highest possible execution speed are gathered in active behavior lists. Besides that, behaviors which are common to both cases and including engineering calculation algorithms are kept actively as a common behavior list.

Case studies

In this section, two illustrative example are given to show how different aspects are defined and implemented for an entity and how a relation to be used to change aspects.

Radar simulation example: The application is related to searching and tracking behaviors in a radar simulation model. Instead of developing two radar models as search radar and a track radar separately, one radar model, which is capable of performing two different modes named as search and tracking, is developed. Essential functions such as detection calculation, rotational movement in a given angle and direction are used at both operating modes. That means the difference between two radar modes is realized in behavior design which manages the functions. In tracking mode, a radar rotation behavior designed to follow a specific target has taken place in the tracking behavior list, while in search mod; the behavior is overridden in the search behavior list as a consistently rotational movement behavior.

In Figure 8a, for two radar behaviors, the behaviors of being active and analysis are root model behaviors. The rotational movements make two radar types different from each other as seen in Figure 8b. In two behavior lists, shown in the figure have presented the logging operation in behavior form as a state definition of behaviors with alternative design. In both modes, the radar processes the event detection and gives detected or not detected decision (Figure 8a) because the behavior is common. Search radar behavior is seen in Figure 8b. It makes angular rotation in defined time steps with *ComputeDuration* time calculator. The search behavior is carried out the rotation between defined angles and any time the limits are exceeded, the turn direction is changed by the function *ChangeDirection*. In track radar behavior, the rotation is triggered by the event *turn*. The event gives a direction to turn and it searches consistently in a given interval keeping the direction given at the center. The turn duration is calculated by the function *StateDuration*. Besides, logging process is being proceeded as a behavior triggered by the tracking behavior. As seen in the example, logging operation as a cross cutting function is achieved by both distributing a state surrounding the function into behaviors and activating as a separate behavior that consists of the function.

Command and control example: A Command and Control (C2) model send weapon target pairs to tactical picture model. The commander is put under command of another commander by constituting a relation between two commanders. It is named as "Commands" relation and it can be used between two commanders. "Commands" relation is designed between two commanders. The first commander is a commander being under command and the other commands. Being under command reports detections that it receives from his/her sensors and the commander that commands collects all the detection coming from the commanders that he/she commands, makes a decision, and gives orders to the commanders or one of the commanders what to do. Reporting detection and waiting for an order is defined as a specific behavior category and it is activated by the commander being under command that is located on right side of the relation (Commander-R-0 and Commander-R-1). The behaviors consisting of whole commander roles such as engagement, target

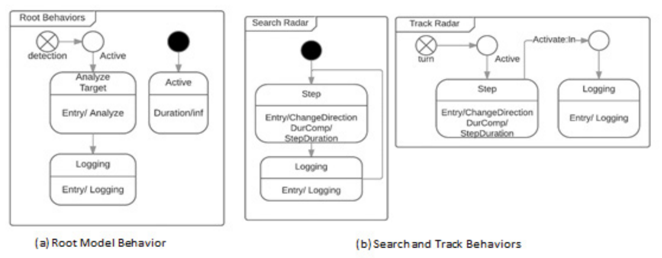


Figure 8: Behavior hierarchy. (a) Root model behavior; (b) Search and track behaviors

evaluation are undertaken by the commander that commands and it is located on left side of the relation Commands. While the commander being commanded to make its behavior list shorten as enough as to send detection and to implement what the order wants it to do deactivates rest of the behavior lists. The commanding commander undertakes all behaviors necessary to manage a set of sub-commanders and to achieve whole requirements of a commander.

From this point of view, in a scenario, different instances of the same entity may have different behavior categories, in other words, different behavioral semantics. In Figure 9, a C2 structure is established between three commander entities. Commander-C commands Commander-R-0 and Commander-R-1. The relation *Commands* is defined for this purpose and the commander that commands is located on left side of the relation and also a relation between Commander-C and Tactical Picture entity is established named as "*Sends Information*". Using the relation, commander sends detection collected from the sensors to tactical picture entity to make them depicted on a map (Figure 9).

As seen in Figure 10, Commander-C entity activates the behavior list named as "DepictDetectionsOnMap" to send detection information to the tactical picture model entity to depict them on a map. The behavior list consists of a set of behaviors for this purpose. Similar way, a commander that has a relation named as "Commands" and it is located on left side of the relation, activates the behavior list named as "CentralC2" that consists of behaviors that are in charge of command and control functions. When the relation is broken, the behavior list named as "*CommandeFsaList*" is activated and the function *Af_SetReportAddress* is executed. Any time establishing and breaking the relation activates related behavior lists and executes related functions.

Notice that the relations, behaviors, and functions are defined by modelers they are not build-in structures. The usage allows modelers to manage dynamically a set of behaviors and functions satisfying dynamically arisen requirements and to handle changing roles in run time.

Conclusion

Combining AdSiF aspect-oriented programming with Model Driven Architecture (MDA)/Model Driven Development (MDD) gives a quite powerful and flexible modeling platform. Aspect-oriented programming approach provides a flexible solution by using behaviors and behavior lists of AdSiF as a design pattern to meet scattered and tangled requirements. With the help of this approach, it is possible to improve simulation and agent modeling whose maintenance and development are quite hard by preventing scattered code of atomic functions which cannot establish any semantic link between each other. AdSiF can be seen as a flexible and user friendly solution based on AOP in terms of providing scattered requirement (cross cutting requirements) with scripting programming language, allocating state

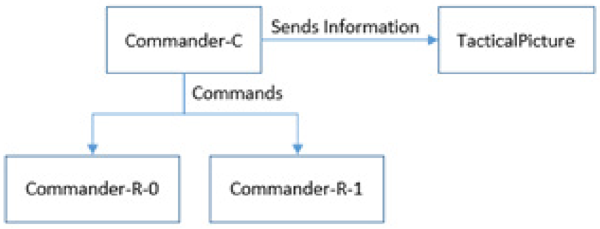
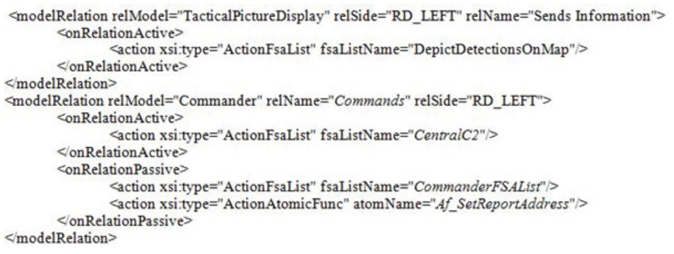


Figure 9: Relations between entities.



```
<modelRelation relModel="TacticalPictureDisplay" relSide="RD_LEFT" relName="Sends Information">
        <onRelationActive>
                <action xsi:type="ActionFsaList" fsaListName="DepictDetectionsOnMap"/>
        </onRelationActive>
</modelRelation>
<modelRelation relModel="Commander" relName="Commands" relSide="RD_LEFT">
        <onRelationActive>
                <action xsi:type="ActionFsaList" fsaListName="CentralC2"/>
        </onRelationActive>
        <onRelationPassive>
                <action xsi:type="ActionFsaList" fsaListName="CommanderFSAList"/>
                <action xsi:type="ActionAtomicFunc" atomName="Af_SetReportAddress"/>
        </onRelationPassive>
</modelRelation>
```

Figure 10: Behavior management by relations.

definitions which covers requirement function to behaviors and using independent behavior forms in behavior lists. Functional extensions includes activation/deactivation run time of a behavior list and software plugin run time provide a considerable flexibility in which aspect programming solution which is transformed design to simulation run time.

The solution gives more than one alternative solution to develop an aspect. These are distributing a state or a behavior that satisfies a scattered requirement or a set of requirement to behaviors and behavior lists, respectively and activating a scattered behavior by temporal relations, phase conditioned, or function conditioned. Indirect behavior activation (more generally behavior phase transition) because of its indirect invocation method, the solution does not intervene function source code and behaviors of simulation entities and/or agents, this support orthogonality criteria in software engineering. Any change on behaviors does not require changing source code or other behaviors.

Introducing relation concept into agent and simulation world makes dynamically change behavioral aspect of the entity in run time possible. Any time a relation established and broken a set of functions and a set of behavior lists (consisting of different aspect properties) are activated and deactivated, respectively. An entity may establish and break many relations many times.

For distributed simulation, some of the time synchronization algorithms are modeled as behaviors. The behaviors are script declarations and they are interested by simulation/agent core engine. The layering and separating entity behaviors and allowing their execution depending on their priorities allow modelers to design their own simulation time management algorithm as a behavior set without making any change on interpreter core.

References

1. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, et al. (1997) Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented Programming – ECOOP, pp: 220–242.
2. Lieberher K, Orleans D, Ovlinger J (2001) Aspect-oriented programming with adaptive methods. Communications of the ACM 44: 39-41.
3. Mendonça NC, Silva CF, Maia IG, Rodrigues MAF, Valente MTO (2008) A loosely coupled aspect language for soa applications. Int J Soft Eng Knowl Eng 18: 243–262.
4. Chibani M, Belattar B, Bourouis A (2014) Practical benefits of aspect-oriented programming paradigm in discrete event simulation. Modelling and Simulation in Engineering.
5. Lucian L, Despi I (2005) Aspect oriented programming challenges. Anale Seria Informatica 2: 65–70.
6. Madeyski L, Szala L (2007) Impact of aspect-oriented programming on software development efficiency and design quality: An empirical study. IET Software 1: 180–187.
7. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, Lucena C, et al. (2005) Modularizing design patterns with aspects. In Proceedings of the 4th International conference on Aspect-oriented software development – AOSD, pp: 3-14.
8. Kersten M, Murphy GC (1999) Atlas: A case study in building a web-based learning environment using aspect-oriented programming. SIGPLAN Notices 34: 340-352.
9. Walker RJ, Baniassad ELA, Murphy GC (1999) An initial assessment of aspect-oriented programming. 21st International Conference on Software Engineering, pp: 120-130.
10. Garcia A, Sant'Anna C, Chavez C, da Silva VT, de Lucena CJP, et al. (2004) Separation of concerns in multi-agent systems: an empirical study. SELMAS, pp: 49–72.
11. Žilvinas V, Cˇaplinskas A (2011) Software engineering paradigm independent design problems, GoF 23 Design Patterns, and Aspect Design. Informatica 22: 289–317.
12. Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20: 476–493.
13. Tsang SL, Clarke S, Baniassad E (2004) An evaluation of aspect-oriented programming for Java-based real-time systems development. Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing.
14. Kumar A, Kumar R, Grover PS (2011). Unified cohesion measures for aspect-oriented systems. Int J Software Engg and Knowledge Engg 21: 143–163.
15. Zambrano A, Gordillo S, Fabry J (2010) A fine grained aspect coordination mechanism. Int J Software Engg and Knowledge Engg 20: 1025–1042.
16. Sirbi K, Prakash JK (2010) Enhancing modularity in aspect-oriented software systems-an emperical study. Int J Comp Sci Engg 2: 2009–2014.
17. Hameed K, Williams R, Smith J (2010) Separation of fault tolerance and non-functional concerns: Aspect oriented patterns and evaluation. Journal of Software Engineering and Applications 3: 303–311.
18. Meslati D (2009) On Aspect J and composition filters : A mapping of concepts. Informatica 20: 555–578.
19. Heidenreich F, Henriksson J, Johannes J, Zschaler S (2009) On language-independent model modularisation. Lecture Notes in Computer Science, pp: 39–82.
20. Anneke K, Jos W, Wim B (2003) MDA explained: The model driven architecture : Practice and promise. Addison Wesley. p. 192.
21. Hocaoglu MF (2005) AdSiF: Agent driven simulation framework. Hunstville Simulation Conference.
22. Hocaoglu MF (2011) EtSiS: Etmen tabanlı Simülasyon Sistemi. In 4 Ulusal Savunma Uygulamaları Modelleme ve Simülasyon Konferansı, USMOS'2011.
23. Fujimoto RM (2000) Parallel and distributed simulation systems. Wiley Series in Parallel & Distributed Computing.