Using LSCs for Scenario Authoring in Tactical Simulators *

Yoram Atir and David Harel†
The Weizmann Institute of Science, Rehovot, Israel 76100

Keywords: Behavioral specification, Live sequence charts (LSCs), Scenario authoring, Tactical simulators, End-user programming

Abstract

The scenario-based approach to programming is adapted for use in tactical simulators, in which highly structured inter-entity scenarios are used for practicing and assessing decision making. The adapted approach, using the LSC language, enables end-users to capture scenario specification directly from the unfolding 3D scene. The specified scenarios are then used both for behavior monitoring and for the orchestration of simulation runs.

1. INTRODUCTION

Computer simulators and the so-called serious games (See, e.g., [1] for the latter), are referred to as "tactical" when their main focus is on decision making, rather than on technical skills. In order to make high level functionalities in the applications susceptible to assessment and control, the simulation is carried out by actors assigning intentions to semi-autonomous agents. The intentions are typically related to other entities, and so the simulation is to a large extent defined by high level interactions between entities. Thus inter-entity scenarios are at the heart of tactical simulations. They are set up in order to practice and assess the interactions of actors with respect to some local doctrine.

The doctrine and the scenarios that embody it are often expressed in a modal, temporal way. For example, "If X occurs, action Y must be initiated in no more than Z seconds". Inter-entity scenarios are typically defined by Subject Matter Experts (SMEs), and may be used both in simulator design and testing, and in simulation runs. The former relates to the expected behavior of (semi)autonomous agents and the latter to the expected behavior of human trainees.

Two challenges in this area are the creation of believable agent behavior and the orchestration of different agents and behaviors into credible, controlled scenarios. A major issue is the gap between the partial and informal specifications given by SMEs in the design stages of the system and the implementation of the behavior, which requires the utilization of formal methods by computer experts. In a larger context, that of the general domain of reactive systems [2], a similar gap was one of the main forces that triggered the introduction of the language of *live sequence charts* (LSC) [3], and the subsequent work on the play-in/play-out methodology and the supporting Play-Engine tool [4, 5].

The play-in technique involves a user-friendly and natural way to play in scenario-based behavior directly from the system's GUI (or some abstract version thereof, such as an object-model diagram), during which LSCs are generated automatically. The play-out technique makes it possible to play out the behavior; that is, to execute the system as constrained by the grand sum of the scenario based information. These ideas are supported in full by the Play-Engine.

In the present paper, we show how to adapt the LSC language and the play-in/play-out techniques, so that they can be applied beneficially to real-time tactical simulation. Among other things, this makes it possible for end-users and SMEs to naturally and intuitively specify LSCs for use in scenario orchestration and behavior monitoring.

The work we set out to do involves dealing with many different issues related to languages, semantics, software engineering and modeling, and the project in general is still in progress. In this paper we outline the suggested methodology, report on an initial (partial) implementation, and discuss some basic issues that arise in the adaptation process. We focus mainly on two of the aspects inherent to real-time simulator applications, which could not have been part of the original LSC language and its play-out technique: First, since simulation is carried out by independent actors, we have augmented the language with the ability to reason about actors. Second, high level scenarios are inherently dependant on low level and domain specific issues, which are often unpredictable at the time of specification. Accordingly, in order to retain the intuitive nature of LSC specification, we have modified the play-out semantics and have established a model and mechanism for considering low level information while abstracting it from the high-level specification. For instance, the model deals with the selection and timing of enabled events to account for low-level dependencies — a non-issue in the original play-out technique.

The paper is organized as follows: Section 2 discusses related work on autonomous behavior and scenario orchestration. Section 3 presents the simulator prototype that we use as a testbed, which is a typical example of the application domain. Section 4 gives an overview of the LSC language and the play-in/play-out methods as they are modified and adapted for use in the suggested setting. Section 5 elaborates on the adaptation issues outlined above, and Section 6 concludes with a discussion of some currently known limitations and a preview of our ongoing and planned work.

*This research was supported in part by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science.

†Part of this author's work carried out during a visit to the School of Informatics at the University of Edinburgh, which was supported by a grant from the EPSRC.

2. RELATED WORK

In general, schemes for applying autonomous behavior must deal with the coordination of objects and behaviors. There are many different approaches to this, in different domains. The vast majority of work that aims for granular and predictable behavior at the level of a single object take an intra-object (and essentially a state-based) approach. This means that an object's behavior depends entirely on its associated action schemes and internal believes. See for example [6, 7]. Some of the research in the intra-object paradigm has targeted the complexity of autonomous behavior implementation, suggesting high level languages. These are often accompanied by tools and methodologies aimed at bridging the gap between high level behavior specification and its implementation. See for example [8, 9, 10]. In related work, such as [11] and [12], it has been noticed that intra-object approaches are not very fitting for highly structured inter-object scenarios. These papers present disembodied high-level state-based agents for orchestrating embodied agents and other orchestrators. In [13, 14, 15] it was also noticed that the state-based representation is inadequate for highly structured inter-object scenarios. Their authors suggest textual-based scenario specification languages, with operators that define temporal relationships between scenarios, and mechanisms to establish the way scenarios are interleaved (e.g., by explicit prioritization). The work in [13, 14, 15] is close to ours in application domain, and in that LSCs may also be viewed as disembodied agents that orchestrate other agents. However, the visual nature of the LSC language makes a rather significant difference, as we shall see.

UML and UML derivatives have been suggested for agent specification in, e.g., [16, 17]. In particular, UML's sequence diagrams (which originated in the telecommunication industry as message sequence charts, MSCs), and specialized versions thereof, have been suggested for specifying agent interactions. Having built on the insufficiently formal basis of the UML standard, and being non-executable, these sequence diagram dialects cannot support a fully operational setting. In the OO domain, formal derivatives of UML's sequence diagrams have been used in testing tools, see, e.g., the TestConductor tool of Rhapsody [18]. In that approach, system events are traced against sequence diagrams in order to detect flaws, mostly with respect to partial order. This method can also be adapted for testing aspects of autonomous behavior during simulator application development.

3. TACTICAL SIMULATOR TESTBED

We now describe our system for illustration. It has characteristics typical of tactical simulators. It is meant to train personnel of so called 'tactical units' in real-time decision making. Tactical units tend to develop a unique type of combat doctrine. This doctrine consists of a set of rules considering where and how to look for threats, how to asses the relative severity of multiple threats, what actions should be taken with regard to these threats, and in what order.

The system is built around a central server, in a star architecture. The stations host end-users, which may be trainees or instructors. Instructors and trainees may operate entities by assigning them intentions, which are typically actions that relate to other objects. For example, entity *A* may be assigned the intention "Move fast to object *B*", where "fast" is a property of the move action. All actions implied by the intention are autonomously performed by the entity. For example, the moving entity *A* may decide on the route to object *B* and may respond autonomously to events like collisions, which may occur on its way to *B*. This 'hardwired' autonomous behavior is usually specified by SMEs earlier, at the prototype development phase, as is the related entities' interface; i.e., its set of possible intensions.

Trainees are associated with a single human entity and are limited to its point of view of the scene. Trainees can only control their assigned entity. Instructors can control every entity, and take any point of view. An intension is associated with an entity by a natural GUI operation. For example, a trainee may instruct its associated entity to move by pointing the mouse at the target and clicking a mouse button. The entities' pose may be toggled, from upright to duck and vice versa, by pressing a keyboard key, etc. Instructors must first associate themselves with an entity, and can then direct it in a similar manner, although they have a richer interface. For example, an instructor can alter the 'health' of an entity, an action that trainees cannot carry out — they can only view its consequences.

Simulation scenarios typically have definite initial and terminal configurations, and are structured to practice specific aspects of the doctrine in predetermined points in place and time. As a consequence, hardwired high level autonomous coordinators cannot be effectively utilized, and inter-entity scenarios are manually orchestrated. This is obviously a problem. Scenarios can be quite complex and may be dynamically interweaved in ways that are hard to predict. In many cases, the orchestration itself is faulty, and important things are missed.

Our LSC examples in the paper will refer to the scene in Figure 1. Entity *Soldier1* (lower left) is shown shooting at entity *Fighter2* (right). The other human entity, on the upper left, is *Fighter3*. The upper barrel-shaped entity is *Stand5*, and the lower one is *Stand4*.

We will refer to instructors and trainees as *actors*. The entities that are directed by actors are referred to as *agents*.

4. THE LSC APPROACH

For fuller details about the scenario-based programming paradigm, the language of live sequence charts (LSCs), the play-in and play-out methods and the supporting Play-Engine tool, we refer the reader to [3, 4, 5]. Here we present only those aspects of the language that are essential to this paper.

Figure 2 shows a *universal LSC* named *Fighter2Action*. This LSC states the required actions of *Fighter2* in response

to being shot at by some soldier. The entities are represented by the vertical lines, called *instance lines*. The two rightmost instance lines represent specific entities, while the left one is *symbolic*, meanin that it can represent any instance of its class, and it does so by dynamically binding to any entity of class *CSoldier* during execution. Events, such as inter-entity messages and self property changes are represented by arrows. Along an instance line, time progresses from top to bottom, so that higher events precede lower ones. Also, in general, the sending of a message precedes its receipt, but in our examples all inter-entity interactions are synchronous, so sending and receiving are simultaneous.

A universal chart has two parts, turning it into a kind of if-then construct. Its top dashed portion, the *prechart*, specifies the scenario, which, if satisfied (i.e., carried out to completion), causes the system to have to also satisfy the chart body, the *main chart*, which is the solid part at the bottom. Thus, such an LSC induces an action-reaction relationship between the scenario appearing in its prechart and the one appearing in the chart body. In our case, The only event in the prechart is the *ShootAt(X1)* intention message from the soldier to *Fighter2*. Possible values for the parameter $X1$ are 1 for slow, and 2 for fast.

Thus, the prechart will be satisfied when some soldier will be associated with an intent to shoot *Fighter2*, at either speed. If and when this happens, the symbolic instance line will bind to the actual shooting soldier, and $X1$ will bind to the requested speed value. This will then cause the following chain of events: *Fighter2* will start shooting back at the soldier and then start a slow moving action in the direction of *Stand4*. After reaching the stand, *Fighter2* will change its pose and duck behind it. This manifests itself as changing the value of property *FightPose* to 1. Eventually, the value of the health property of *Fighter2* will change to 0. This results in a short 'dying' animation, and the entity will no longer be operable.

The scene in Figure 1 was constructed for assessing the soldier's behavior. Fighter2 and Fighter3, as well as the stands, were positioned at the scene building stage. While it was known that some soldier will arrive at the scene, his/her exact identity was not known, which explains why the soldier's instance line is symbolic. While the actions of the soldier are monitored, that is, the events are 'given' to the Play-Engine by the entities themselves, the behavior of the fighters is driven — i.e., orchestrated — by the Play-Engine itself. This is indicated by the "PE" tag associated with *Fighter2*'s activities in the LSC, so that the Play-Engine is responsible for initiating the associated activity. The only event in *Fighter2Action*'s main chart not initiated by the Play-Engine is the *Reached()* message from *Stand4* to *Fighter2*. Obviously the Play-Engine cannot initiate this event, but it monitors it for establishing that ducking (and dying) will not occur before the stand is reached.

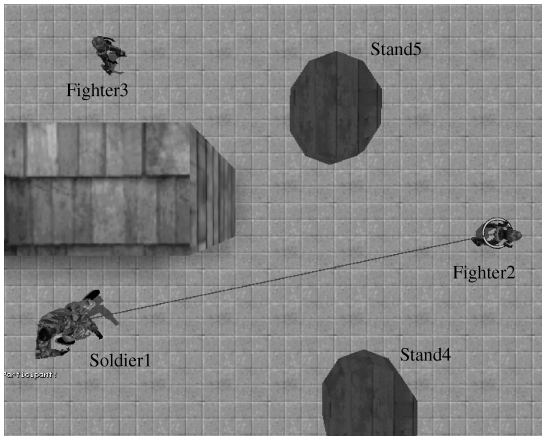

Figure 1. An example of a scene

Note that the only temporal specification of events in the LSC is the partial order between them (given by the top-to-bottom and send-before-receive rules). For instance, nothing is said about the exact time between ducking and dying. This information emerges dynamically from other LSCs or from actual properties of the simulation, as we shall see later. Note also that the LSC specifies only 'skeletal' behavior. For instance, it says nothing about things that *Fighter2* might do between shooting back at the soldier and moving to *Stand4*. It only states that *Fighter2* must eventually decide to move to *Stand4*, and not duck (or dye...) before reaching it. Such weaved behaviors may be expressed in a combination of different LSCs, or ignored altogether. The possibility of underspecifying the behavior while effectively concentrating on key aspects of it conforms with the intuitive way scenarios are conceived by SMEs; it is actually a key feature of our approach, making it amenable for end-user programming.

4.1 Play-in: Programming scenarios

The Play-Engine is currently a stand-alone application, with its own dedicated GUI. In it, LSCs are depicted during play-in, and can be visually traced during play-out. It also allows for a specific GUI of the application to be present and

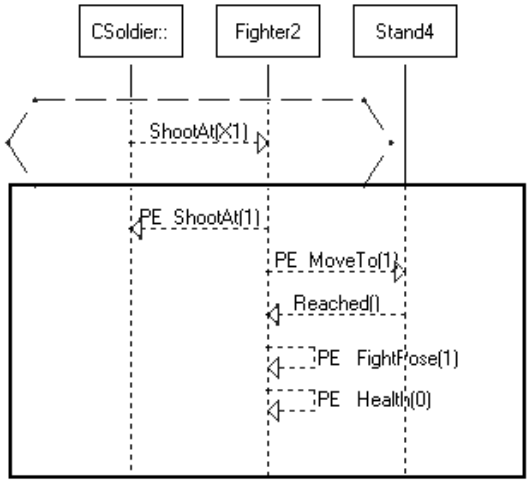

Figure 2. *Fighter2Action*: A universal LSC stating the required actions of *Fighter2*

be animated during play-in and play-out.

The LSC *Fighter2Action* was 'programmed' in much the same way its described scenario unfolds during simulation run, using the actual simulation scene given in Figure 1. A soldier entity was brought to its initial place and the Play-Engine's mode of operation was set to play-in a newly opened LSC. The operator then assigned the soldier the intention to shoot at *Fighter2* in one of the ways it would have done so in actual simulation run. This had two simultaneous effects: the soldier entity started the shooting action and the *ShootAt(X1)* message was depicted in the LSC's prechart. The shooting speed initially had a concrete value matching the actual operator's action, but it was later changed to be symbolic. The operator then moved to the main chart's region in the GUI, and chose *Fighter2* as a default controlled entity. As before, from here on the LSC directly reflects the actions of the operator and the related happenings in the scene. For example, After choosing *Stand4* as a target of the movement, the operator waited for the entity to reach it. As this happened, the entity stopped moving and the *Reached()* message appeared in the LSC.

During this specification process, the soldier's instance line was bound to the exact entity that was brought to play. It was changed to be symbolic after the specification process was completed. Similarly, the Play-Engine was assigned to be the initiator of events at the end of the play-in process. Note that play-in does not have to take place during a (stub) simulation run. It is possible to play-in on a still scene, as well as on an OMD of it. In fact, we found it convenient to occasionally stop a scene or exit the play-in mode, for the sake of 'manual' adjustments, or in order to avoid playing-in redundant events.

The LSCs shown here are very simple, to ease the exposition and keep the paper short. Producing them involved mainly acting in the simulation scene, with very little interaction with the Play-Engine's GUI. However, as LSCs grow in complexity and include constructs other than events (e.g., sub-charts, assignments, etc.) their creation requires more interaction with the Play-Engine's GUI.

4.2 Play-out: Simulation run authoring

Play-out is the process of executing an LSC specification. In our context, it should be viewed as the method by which the Play-Engine becomes an actor, constantly monitoring the behavior of agents that act on its behalf or on the behalf of other actors, and directing agents as faithfully as it can, according to the grand sum of (relevant) LSCs. In some occasions, the Play-Engine might announce that an LSC has been interrupted, or even violated; i.e., that a scenario has evolved in a way forbidden by some LSC.

Here we only sketch the play-out method as applied to the tactical simulator setting, and describe its effects with respect to the scene in Figure 1, the LSC *Fighter2Action* and two additional LSCs, *Fighter3Action* and *RequiredResponse*, shown in Figures 3 and 4. Some of the issues concerning the implementation of the method in our setting are presented in the next section. For a detailed description of the (original) play-out method, please refer to [4].

The LSC *Fighter3Action* states that as a consequence of a soldier shooting at *Fighter2*, *Fighter3* should start moving to *Stand5*. Also, *Fighter3* should eventually shoot at the soldier and *Fighter2*'s health should drop to 0, but both of these events should occur only after *Fighter3* has reached *Stand5*. This constraint is due to the *SYNC* construct appearing underneath the *Reached()* message. LSC *Fighter3Action* is shown mainly to demonstrate how eventual behavior can be dynamically defined by several different LSCs, which refine and constrain each other.

Assume that the initial configuration is valid, and that sometime during the run *Soldier1* arrives at the scene and issues a *ShootAt(2)* event at *Fighter2*. The play-out mechanism unifies this event with the messages in the precharts of LSCs *Fighter2Action* and *Fighter3Action*, the messages are propagated and both precharts become satisfied. By the main chart of *Fighter2Action*, the Play-Engine then initiates a *ShootAt(1)* at the soldier, then a *MoveTo(1)* to *Stand4*. It waits for the arrival of a *Reached()* event back from the simulator and then issues a property change of *Fighter2*'s Fighting pose. In parallel, by the main-chart of *Fighter3Action*, the Play-Engine initiates a *moveTo(1)* of *Fighter3* to *Stand5* and waits for a *Reached()* acknowledgement. Now comes the interesting part: the *Health(0)* event for *Fighter2* will not be issued before *Fighter3* Reaches *Stand5*. This is because the play-out mechanism unifies the *Health(0)* events in both LSCs, and due to the *SYNC* construct in *Fighter3Action*.

Eventually, *Health(0)* will be issued for *Fighter2*, and LSC *Figther2Action* will close. But according to *Fighter3Action*, *Fighter3* still has to shoot at the soldier. This may be 'physically' impossible, since even from *Fighter3*'s new position, near *Stand5*, the soldier may still be obscured by the wall. So the LSC *Fighter3Action* is left pending until something makes the soldier progress to be in sight of *Fighter3*. What can make him/her do that? Recall that the soldier is human-controlled and hence his/her behavior should be monitored; it makes no sense for the Play-Engine to force the required

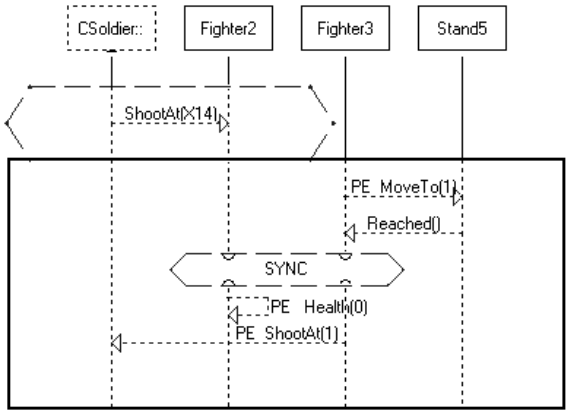

Figure 3. *Fighter3Action*: A universal LSC giving the required actions of *Fighter3*

progress.

Assume that according to the soldier's doctrine, he/she should approach *Stand4* after *Fighter2* ducks behind it. This is stated in LSC *RequiredResponse*. Using a *forbidden condition* this LSC also states that the soldier should start this movement in no more than 2 time units. In the prechart, when *Fighter2* ducks, the current time is captured in the variable T. The scope of the forbidden condition ($Time > T + 2$) is the main chart, so that the condition is monitored from the moment the prechart is satisfied. Since the condition is hot, if the main chart is not exit by the time it becomes true, a violation occurs and the Play-Engine's GUI issues a matching notification. This is one way to handle a deviation from a planned scenario. In general, different paths a scenario might take can be handled by conditional constructs of the LSC language, or by utilizing the basic pre-chart/main-chart mechanism, i.e., by devising LSCs that will be activated or closed, when the scenario path renders them relevant or irrelevant, respectively.

5. ADAPTATION ISSUES

The biggest challenges in our proposed adaptation of the LSC approach to real-time tactical simulators are related to the play-out method. The original method defines a clear cut between the system and its environment (e.g., external objects); see [4]. Every action in the system itself is determined by the Play-Engine according to the LSC specification. Although objects with a graphical representation can be animated, their animation is really under the control of the Play-Engine. Accordingly, every piece of information of interest in the system's behavior has to be embedded in the LSC specification itself.

The original play-out method thus makes two important assumptions: (1) a system object does nothing unless it is told so by the Play-Engine, and (2) a system object will always perform Play-Engine requests immediately.

Obviously, these assumptions do not hold in our setting. As to (1), several actors may direct a single entity. This raises several issues. Most prominently, the Play-Engine has to know who is, or was, responsible for the initiation of events. To deal with this we associate each event with a property that identifies its initiating actor. This is essential for collaborative scenario authoring. In our current implementation of the proposed approach, LSCs explicitly identify only the Play-Engine and 'all other' actors, as can be seen in our example LSCs. We are in the process of considering a more powerful approach.

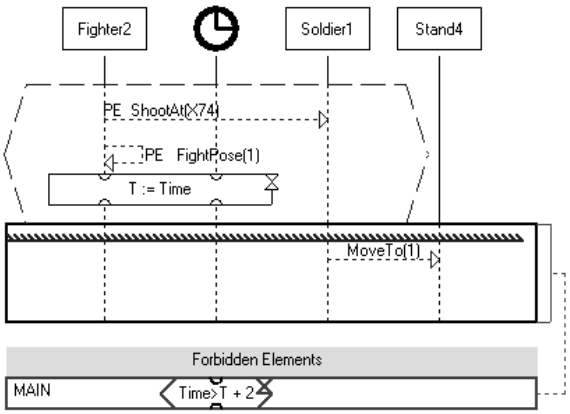

Figure 4. *RequiredResponse*: The response of the soldier to *Fighter2* hiding

As to assumption (2), apart from not knowing the future actions of other actors, the Play-Engine lacks additional information. For example, being a separate generic component, it does not have the scene's full 3D information. As a consequence, it cannot tell if, e.g., *Fighter3* can see the soldier, and *Fighter3* cannot initiate fire towards targets it cannot see. In the simulator, such impossible requests are simply discarded. Hence, the original play-out semantics will probably yield an incorrect scenario, since the Play-Engine would have issued a premature *ShootAt* request.[1]

A major challenge in applying the scenario-based approach to the simulator setting is how to effectively integrate the approach while retaining its power and generality. We would like to make it possible for users to correctly specify and apply high-level 'skeletal' LSCs while ignoring low level, domain specific issues.

To this end, we have devised a Play-Engine proxy, which is a low level agent of the Play-Engine in the simulator domain. It is embedded in the prototype's code in the sense that it publishes an interface for use in different places in the code, and it spawns internal activities that are specifically for the sake of the Play-Engine. On the other side, the proxy has a generic interface to the Play-Engine, and is implemented with XML and RPCs.

Among other things, the proxy is responsible for maintaining the view of the scene synchronized between the simulator and the Play-Engine. This is not at all trivial, and is dependant on low level simulator specification. So, for example, the proxy discards redundant scene messages using deep knowledge of local specification and implementation. The proxy is also responsible for extracting and translating implicit information in the scene, for the sake of modeling. For example, the *Reached()* event is only implicit in the simulator, but is used extensively in scenario specification. The proxy also cooperates with the Play-Engine to initiate events in a locally consistent manner. For example, it initiates a dependent event, such as *ShootAt*, only if and when it is possible according to up-to-date 3D information and low-level local specification.

This last-listed point in particular relates to the way the play-out semantics is enforced by the interplay between the Play-Engine and its proxy. The general scheme is as follows: When an event reaches the Play-Engine from the proxy, the Play-Engine updates its current configuration by unifying and

[1] Actually, the approach can be made to support some kind of 'can-see' functionality (expose a function or an event). However, it can be shown that this does not necessarily enable correct or convenient modeling, due to reasons that are outside of scope of this paper.

propagating LSC events, spawning new LSC live copies, etc. As a result, a new set of events is considered by the Play-Engine as enabled for initiation according to the LSC specification. This set, however, may include events that are not 'physically' enabled, and is thus sent to the proxy. The proxy tries to initiate one of the events in the set in a heuristic manner, considering the likely urgency of events (dependent events are considered more urgent) and the cost of testing the possibility of their initiation. This process continues until initiation succeeds, or until a new set of events arrives from the Play-Engine.

We expect the low level behavior of the proxy to be highly dependant on the specific simulator application. Our main effort in this respect is to devise a highly generalized integration scheme. A step in this direction is the classification of events for instantiation by the proxy according to the way they are considered in play-out. For example, the *Reached()* event is classified as an event that the Play-Engine does not initiate. In play-in, the Play-Engine cannot be declared as an actor for events of this class.

6. CONCLUSION

We suggest that the scenario-based approach to programming, as exemplified by the LSC language and the play-in/out methodology, can be adapted and adopted as a powerful tool for end-user's programming and real-time authoring of tactical scenarios. The main advantage of the approach is the ability to express inter-entity scenarios in a natural and intuitive manner. For effective RT programming, the approach leverages on, and may cooperate with, other programming models. For example, LSCs may author the behavior of entities that are directed by state machines which, in particular, scale better than LSCs computationally. LSCs do not scale well compared to state-machines since the way we have set things up so far is highly centralized. All LSCs in a given configuration have to be scanned for each incoming event. This may become a problem in large scale simulations, as the complexity of LSC specification increases. The cooperation of several Play-Engine's is a possible solution, which should be accompanied with an efficient distribution scheme for the LSC specification, something our group has been working on for some time.

Although our initial results from implementing parts of the proposal are promising, usage tests by end-users have not yet been preformed on a sufficient scale. Also, some elements of the LSC language have not been dealt with yet. We plan to extend the language in order to make the approach more suitable for real world simulator applications. For some useful extension a suitable formal basis exists, and what is required is careful adaptation and implementation. Other extensions require significant additional research.

Acknowledgements We would like to thank Gili Hess for her cooperation and for her work on the simulator prototype.

REFERENCES

[1] Serious Games Initiative, http://www.seriousgames.org.

[2] D. Harel and A. Pnueli, "On the Development of Reactive Systems", *Logics and Models of Concurrent Systems (K. R. Apt, ed.)*, NATO ASI Series, F:13, 1985, 477–498, Springer-Verlag.

[3] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design*, 19:1 (2001), 45–80.

[4] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.

[5] D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach", *Software and System Modeling* 2 (2003), 82–107.

[6] H. Noser and D. Thalmann, "Sensor-based Synthetic Actors in a Tennis Game Simulation", *The Visual Computer*, 14:4 (1998), 193–205.

[7] E. Norling and L. Sonenberg, "Creating Interactive Characters with BDI Agents", *Proc. of the Australian Workshop on Interactive Entertainment IE2004*, 2004.

[8] S. Donikian, "HPTS: A Behaviour Modelling Language for Autonomous Agents", *Proc. 5th Int. Conf. on Autonomous agents*, 2001, 401–408, ACM Press.

[9] M. Kallmann and D. Thalmann, "A Behavioral Interface to Simulate Agent-Object Interactions in Real Time", *Proc. Computer Animation*, 1999, 138–146, IEEE Computer Society.

[10] T. Ishida, "Q: A Scenario Description Language for Interactive Agents", *Computer*, 35:11 (2002), 42–47, IEEE Computer Society Press.

[11] O. Ahmad, J. Cremer, S. Hansen, J. Kearney, and P. Willemsen, "Hierarchical, Concurrent State Machines for Behavior Modeling and Scenario Control", *Prof. Conference on AI, Planning, and Simulation in High Autonomy Systems*, 1994.

[12] J. Cremer, J. Kearney and P. Willemsen, "Directable Behavior Models for Virtual Driving Scenarios", *Trans. Society for Computer Simulation*, 14:2 (1997), 87–96.

[13] J. Kearney, P. Willemsen, S. Donikian, and F. Devillers, "Scenario Languages for Driving Simulation", *Proc. Driving Simulation Conf.* 1999, 377–393.

[14] F. Devillers and S. Donikian, "A scenario Language to Orchestrate Virtual World Evolution", *Proc. ACM SIGGRAPH/Eurographics Symp. on Computer Animation*, 2003, 265–275, Eurographics Association.

[15] P. Willemsen, "Behavior and Scenario Modeling for Real-time Virtual Environments", PhD thesis, Department of Computer Science, University of Iowa, 2000.

[16] M. Papasimeon and C. Heinze, "Extending the UML for Designing Jack Agents.", *Proc. Australian Software Engineering Conference*, 2001, 89–97.

[17] B. Bauer J. P. Müller and J. Odell, "Agent UML: A Formalism for Specifying Multiagent Software Systems.", *Int. J. Software Engineering and Knowledge Engineering*, 11:3 (2001), 207–230.

[18] I-Logix Inc., http://www.ilogix.com.