PRESTO Script: scripting for serious games

Paolo Busetta [1] and **Marco Robol** [2] and **Paolo Calanca** [3] and **Paolo Giorgini** [4]

Abstract. PRESTO, a recently concluded R&D project, has developed a tool suite for the creation of behavioral AI for serious games, with a focus on 3D games for emergency training. This paper focuses on its scripting language, meant to ease development of games by instructional designers and to enable run-time control by trainers supervising a training session. The purpose of the scripting language is to control the choreography of NPCs in the game according to the situations occurring in the game and the trainer's decisions. Scripts describe graphs of potential stories, whose progress and ramifications are determined by events occurring in the game, including trainer's choices. We provide an overview of PRESTO Script, its semantics and its implementation, and compare it with a few commercial and research languages with similar purposes for games and other uses.

1 Introduction

Serious Games based on Virtual Reality (VR) are gaining more and more interest as an effective training alternative to real-life emergency simulations, as demonstrated by the number of products appearing on the market. However, they need to be easily adapted to specific training purposes, possibly by the trainers themselves or by domain experts and so-called *instructional designers*. This requirement of continuous and rapid customization, negligible (if not even considered harmful from a business perspective) in entertainment video games, complicates the development of serious games, thus limiting their diffusion.

PRESTO (Plausible Representation of Emergency Scenarios for Training Operations) [3, 5] was an industrial R&D project,[5] led by Delta Informatica Spa, based in Trento (Italy), and involving the local University and other research centers. The idea at its basis was to create a game-independent all-round development environment for NPC (Non-Player Character)'s behaviors. To ease the diffusion of serious games in non-military domains, PRESTO proposes to reduce their development cost by enabling NPC's model reusability and creating an efficient modeling environment designed for instructional designers and trainers rather than software engineers.

One of the most important developments of PRESTO has been its game-level scripting language, called PRESTO Script. As described later, PRESTO Script is best suited to describe a choreography of behaviors of entities within the game. This is done by describing *stories* as graphs of *scenes* that replace each other with the unfolding of events within the game, and by allowing an arbitrary number of scripts to run in parallel. Scenes simply give goals to NPCs and perform limited manipulations of the virtual environment. The scripting system allows an end-user (supposed to be a game supervisor, in particular a trainer overlooking a training session where one or more trainees play with a PRESTO-enhanced serious game) to take decisions in real-time, in particular about how to let the game proceed at critical points and to restore a previous state of the game e.g. to experience alternative stories from a common beginning. PRESTO Script is currently used by Delta Informatica for emergency training as a commercial service.[6]

This article shortly describes the PRESTO environment, which provides the foundations for its scripting language, focusing on its semantic data (Sec. 2) and the framework for NPC behaviors (Sec. 3). PRESTO Script is discussed in detail in Sec. 4. Sec. 5 identifies a few comparable commercial and research languages and highlights the specificities of PRESTO Script.

2 Virtual environment semantics

To agents controlling NPCs' behaviors and to scripts overseeing parts of the entire game, PRESTO offers abstractions of the virtual environment that are enriched with data that include (but exceed) graphic appearance, instantaneous movements in space, physics and other properties managed by the game engine. Further, these abstraction are meant to be understandable both by domain experts, including trainers, and by developers.

PRESTO distinguishes *entities*, representing physical objects or phenomena in the game and typically corresponding to individual or groups of graphical entities, from *locations*, which are geometrical areas of relevance to reasoning. Differently from entities, locations do not have any representation in the 3D world; they are configured and known only within PRESTO. Even if they are not forced to coincide with anything in the game, the locations should identify semantically meaningful spaces of the 3D world (such as places, rooms, navigation areas...) required by models and scripts to take decisions; e.g., a hospital's "bedroom" and a "operating theater" are both rooms but with well distinguished functionality; a "safe zone" for a fire procedure may simply be a nondescript corner of a parking area.

PRESTO supports three forms of semantic information, referring to specific entities or locations or overall states of the game. These are: (i) classifications, (ii) qualities, (iii) situations.

Entities and locations are classified with respect to an ontology [7, 2], defined with W3C's OWL. A game-independent top-level classification exists but it can be extended per game. Association of game objects to ontological classes is typically done at game's bootstrap: the PRESTO integration layer scans all entities and builds a

[1] Delta Informatica Spa, Italy, email: paolo.busetta@deltainformatica.eu
[2] University of Trento, Italy, email: marco.robol@unitn.it
[3] Delta Informatica Spa, Italy, email: paolo.calanca@deltainformatica.eu
[4] University of Trento, Italy, email: paolo.giorgini@unitn.it

This research has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 699306 - SESAR Joint Undertaking, and under agreement No 653642 - VisiOn.
[5] PRESTO was partially funded by a grant of the Provincia Autonoma di Trento (PAT), Italy.

[6] See the videos at http://lab.deltainformatica.eu/Video for examples (Dec. 2016).

classification table. This classification information is then provided to agents as part of perception data, together with geometrical ones. [7] discusses the facilities that have been built to semi-automatically classify the objects in the rich XVR library for emergency training, while a trivial hand-made procedure has been followed for PUG, which has a limited variety of objects. Ontological queries are supported at run-time, so it is possible to write conditions such as "is entity type E a human character?" or "is location type L an office?".

In addition to being classified and having a limited set of universal properties (mostly geometrical, e.g. position, size, rotation), entities and locations can have ontological properties called *qualities*, containing any form of data needed by agents for their reasoning. Qualities are defined in the ontology (using the owl:ObjectProperty and owl:DataProperty types) and are assigned to entities and locations at run-time. They include states (such as posture of avatars, open/close position of doors), attributes enabling actions (such as "crossable" for doors and any object that needs to be opened during navigation), relationships between entities and locations, for example spatial ("isInside" or "hasInside"), and coordination information (such as "engaging" entities [16]). Qualities are also exploited, by DICE (described later) and the PRESTO integration layer, to reduce the need for intention recognition, by publishing selected goals and on-going actions in the "isPerforming" quality of the related NPC.

Situations represent high level information about the state of the virtual world shared among all PRESTO components; see [11] for a detailed discussion. Technically, a situation is a tuple representing a predicate, i.e. "name (parameters)", whose truth value can be asserted by any PRESTO component or automatically monitored by the so-called *situation engine* while the game evolves. Further, the situation engine makes all true situations visible to all components, thus implementing a form of blackboard system [6]. Situations can be used to simplify reasoning within the agents (e.g. "Fire-In-Room (Room-3)" may represent an accident situation without the need for agents to infer it from perception), to maintain shared cognitive information (e.g. "Firefighter-team-engaged (Team-1, Room-3)" may represent who is doing what and where), and to support the overall game's choreography by coordinating concurrently running scripts (e.g. "Fire-Handling-Procedure-Active()" may represent which section of a training script is currently in progress).

Automatic monitoring requires an expression associated to the situation. The situation language allows to write boolean expressions that contain symbols, parameters names, ontological classes and qualities (see [11] for details). A symbol is defined by specifying filtering criteria used to match entities or locations at run-time, similarly to script symbols described later. The boolean expression is composed by predicates, which allow the comparison between entities or locations properties and base values, e.g.: fire//intensity $>$ $3.5f$, where the // operator reads the quality "intensity" associated with the symbol "fire". A predicate can also test a relation among entities or locations using a relational quality and the operator **contains** as in the following: fire//isInside **contains** (reception). In addition to usual boolean operators, the language supports quantifiers (existential, universal and counting), but their application is restricted to single predicates and refers to entities or locations. For example:

(all firefighter) //isInside **contains (any** firefighter_truck)

AND **count**(firefighter) > 0 AND **count**(firefighter_truck) > 0

Counting quantifiers can also be used to count how many entities or locations have a specific relation with a specific one, for example: **count** (reception//hasInside) $>= 1$

At run-time, agents and scripts invoke the situation engine to assert or retract a situation or to ask to monitor its truth, specifying the values of its parameters. The engine notifies its subscribers of changes to any situation and to its associated symbols. The engine adopts efficient, game-specific algorithms to decide when to evaluate the expressions of monitored situations, e.g., by installing listeners on the objects specified as parameters that are invoked when their qualities change. Automatically monitored situations are suited to capture dynamic configurations of the virtual world, while asserting or retracting situation from code is appropriate for coordinating agents and game-controlling scripts.

3 Virtual actors

PRESTO relies on autonomous agents to animate NPCs and other entities requiring some form of behavior. This section provides an overview of PRESTO's most complex framework, sufficient to understand how PRESTO Script interfaces NPCs via their meta-data (most importantly, agent types and roles), which are supported also by other, simpler agent frameworks, and to appreciate the rich behaviors that may be obtained by exploiting all capabilities offered by PRESTO.

DICE [7, 3] is a BDI (Belief-Desire-Intention) [1, 14] agent framework built on top of JACK[7] [4] for controlling NPCs of a game encapsulated by PRESTO, which offers services such as situation awareness and game-independent action execution. DICE has been inspired by another JACK extension, CoJACK [15, 8], specifically addressing cognitive simulation and in particular variability of behaviors.

Typically, a DICE agent controls one NPC. DICE supports the needs of game-level scripting done with PRESTO Script. Ideally, DICE models (by their BDI nature) should look after agent-specific tactics to achieve goals, while scripting should represent multi-agent strategies. In practice, things are rarely so clear-cut; nothing prevents the development of fully autonomous strategical agents as well as of scripts that implement detailed procedures at the tactical level.

DICE adopts a simplistic cognitive model in which an agent pursues at most one high-level goal at the time (which may have been forced by a game script) and urgently deals with at most one interruption (such as an alarming situation) that must be fully handled before continuing with the high-level goal. With respect to JACK, DICE greatly simplifies changing goals, which is required to support sudden changes of mind (possibly in reaction to events) and to promptly act when receiving commands from the outside world, including scripts.

DICE provides various types of meta-data and introspection facilities on top of JACK that enable model composition, plan interpretation, emotional influences, and game-level scripting. Introspection is available as an API for user-written meta-level models. Further, goals and plans can be annotated with one or more ontological class, which are automatically published as qualities of a NPC when its controlling agent activates an instance (i.e. it starts pursuing a certain goal or intends to execute a certain plan) and unpublished when deactivated. This specific meta-data is exploited, among other things, for recognition of activities by other agents, by the situation engine, to support multi-agent coordination, all in a model-neutral way since the ontology is written from the perspective of an external observer with no knowledge of the inner workings of the different types of agents (e.g., a generic "operating device D" class may be used to an-

[7] http://www.aosgrp.com/products/jack/, last accessed Jan. 2017

notate very different plans or goals applied by different agent models when performing any task that requires using a device of type D).

DICE supports two interpreted languages aimed at end-user development for the rapid creation of ad-hoc procedures rather than for complex, reusable modeling, which is better left to native JACK programming. The first one is a very simple textual language that allows the concatenation of goals to be achieved in sequence, with optional duration and non-conditional loops. It allows writing simple linear procedures such as (using a simplified syntax for readability) "go to place P; repeat (do something for 5 secs; say hello)" as a single string that can be given to an agent e.g. by a PRESTO Script.

The second language is called DICE Part, where "part" is a term taken from theater to refer to the text that an actor has to interpret in a play. A DICE Part file, syntactically represented in XML but more easily edited with an ad-hoc GUI designed for end-users [12], implements a complete behavioral model.

4 Game-level Scripting

PRESTO Script is, in a sense, the tip of the PRESTO iceberg, since it provides end-users with an efficient way to develop parts of game logic (or even the entire logic in open-world virtual realities or simulations, as in the case of XVR) and multi-agent strategies. It exploits all PRESTO and DICE metadata, including roles, situations and ontologies. Scripts, possibly custom-made for specific training objectives, have to be executed within an existing virtual environment, exploiting available NPC models (which themselves may have been customized with DICE Parts, described previously) as well as any pre-built script handling e.g. common cases or team strategies.

The PRESTO Script suite includes a high-level language, a visual editor, an engine, and a controller. The engine and the controller are tools conceived to be used only at run-time; the engine interfaces with PRESTO and interprets the scripts, while the controller provides a UI to a supervisor that allows to start and stop scripts and to take choices interactively. The editor is an off-line tool that supports a specialist in the development of scripts. Editor and controller can run on different machines than those with the game engine, the PRESTO Script Engine and the rest of PRESTO, enabling distributed development and remote game control. The scripting language permits the description of a possible story as a graph of scenes where goals are delegated to agents and commands are submitted to the virtual reality engine; a walk in the graph identifies the unfolding of a story, which happens according to the situations occurring in the game (Sec. 2) as well as interactive choices and timers, independently of the execution and outcomes of the commands and goals delegated at each step. Thus, the engine acts as the director of a choreography performed by entities in the game (which includes the human player acting within the environment) rather than a mere executor of a workflow; further, by means of the controller UI, it allows a supervisor (e.g. a trainer) to run an arbitrary number of scripts concurrently, to terminate any of them at any time and, if supported by the underlying graphical engine, to restore the state of the virtual environment as it was at predefined points of the scripts. The ability of choosing scripts, selecting alternative paths and returning to previous states gives full control on training sessions and enables the interactive exploration of alternative stories within a single virtual environment.[8]

[8] One of the videos at http://lab.deltainformatica.eu/Video illustrates a full example of script editing and execution (Dec. 2016).

4.1 Overview of the language

Technically, PRESTO Script is a simple event-driven language for submitting goals to NPCs, asserting situations or changing properties of game entities according to progresses in the game or choices of a director (typically an instructor). Its syntax is XML-based (similarly to the DICE Part language). A script contains a set of *prerequisites* and a directed graph of *scenes* connected by *events*; the graph admits arbitrary connections, including cycles.

The prerequisite section has the double purpose of (i) checking that the virtual environment contains the objects (entities and locations) expected by the script and (ii) binding these objects to symbols usable from scenes. Similarly to situation language (Sec. 2), a symbol contains the criteria to be matched against objects (name, classification, agent role and agent type as well as position relative to the items bound to other symbols); further, it can impose cardinality constraints (none, one, n, at least or no more than n) and a boolean expression on the qualities of the bound entities At run-time, a failure in binding a symbol (i.e. in finding the required number of entities matching its criteria and satisfying its expression) implies that a prerequisite is not satisfied, so the engine will report an error and will not run the script. This version of the language does not support user defined variables other than symbols.

Scenes, i.e. the nodes of a script's graph, can be of different types. There are two command-executing scene types. The first is called "plain" and contains one or more commands to be applied when the node is reached. A command can be a property change (which is assumed to happen instantaneously), a goal delegated to an agent (without waiting for the latter to process it), the creation or removal of entities, the assertion of the truth value of situations. The symbols declared in the prerequisite section are used to specify the performers and parameters of commands. The second command-executing scene type, called "subscript", invokes another script. Parameters can be passed; their names correspond to the subscript's symbols, whose matching criteria and constraints are used to check the values passed as input rather than to search objects in the virtual environment. Subscripts can be invoked synchronously, i.e. as if they conceptually expanded the calling graph, or asynchronously, i.e. to run concurrently with the caller. A script execution terminates when reaching a scene of type "success" or "fail"; when reached within a synchronous subscript, these nodes generate an event to be handled by the caller to continue its own execution.

Events label the edges between scenes. They include the expiration of timers, the conclusion of goals submitted by the originating nodes, the truth value of situations (true or false), and user choices. At run-time, the engine examines each edge outgoing from a node that has just been processed and registers appropriate event-catching listeners. In the case of situations, the situation engine is invoked to query their current truth values and, if they do not match what is required by the events, to subscribe to change notifications. Choices are delegated to the controller UI, which will show their destination scenes to the user and will notify the engine when one is selected.

The way multiple edges originating from a node are treated at run-time when one of their events occur depend on the node's type. Those from a command-executing scene (plain and subscript, discussed above) are considered as the start of alternative branches of a story (similarly to Finite State Machines). This means that, at run-time, a walk in the graph will follow the edge whose event is captured first by the engine, discarding all other paths; if the latter include a choice edge, the controller UI is notified to remove its destination among those allowed.

Parallel walks are supported by "fork" scenes. Forks do not submit commands; simply, they visually represent a point from where concurrent paths start. At run-time, all edges from a fork will be followed in the order their respective events occur; this means that all their destination nodes will eventually be processed, possibly at very different times. The opposite of a fork is a "join" scene, of which there are two specializations. Also this type of scene does not submit commands; it is considered ready for processing when reached by any of its incoming edges ("join-any") or by all of them, possibly at very different times ("join-all"). Processing consists of blocking walks on all still active paths (if any) that started from the fork specified in the join itself. The edges outgoing from a join are treated as alternatives, as in command-executing scenes. Observe that a parallel walk started by a fork and closed by a join-any can be used to implement activities such as parallel searches to be stopped as soon as a result is found, while a fork closed by a join-all is appropriate e.g. for the distribution of equally important tasks among members of a team and the synchronization on their conclusion.

As an example, the left side of Figure 1 shows a snapshot of the graph view of an example of PRESTO Script as shown by the script editor. A fork node (yellow square, fork symbol downward) is the starting point of three parallel paths: the rightmost one starts a subscript "firefighter_control" when the situation "started_fire" becomes true, the central one starts the subscript "fire_control" immediately (0-seconds timeout), the leftmost one is followed if and when the director decides to do so, in which case a goal (not visible from the diagram) is given to an NPC by the orange scene. The join-all point (green square, fork symbol upward) allows further progress only when all its incoming paths have been traversed; as an alternative option, not shown here, a join-any would be satisfied by the first path traversed, causing the others to be immediately abandoned.

It is worth highlighting that symbols and sub-scripting within the language, in addition to the PRESTO abstractions represented by roles and qualities, enable the development of scripts that can be reused multiple times and in different environments. Script-asserted situations can be used to coordinate scripts running in parallel, in addition to abstract them from the specificity of the environment and game state.

Scenes can be marked as "checkpoints". When a checkpoint scene is reached, if supported by the virtual environment, the engine takes a snapshot of the state of the world (including goals being performed by the agents) and the controller UI is notified, allowing the user to ask to restore the state of the environment and of the running scripts at a later time.

4.2 Language Semantics

A preliminary formal semantics for PRESTO Script is described in [11]. Importantly, scripts should not be analyzed in isolation but – since they can be arranged in hierarchies of scripts and subscripts and coordinated via shared situations – as a set of cooperating procedures.

Operationally, the algorithm at the core of the script engine maintains a queue of incoming events and the list of instances of edges from all currently active scripts waiting for events to occur and manipulates the tuple space of situations (maintained by the situation engine described in Sec. 2). Note that the same script may be running in more than one instance simultaneously, e.g. as subscript invoked by two other scripts; script instances may have different bindings of their symbols, and their edges are differentiated accordingly.

The following is a simplified outline of the algorithm. Initialization (not shown) happens when the first script is started, e.g. interactively (see the discussion on implementation later). When a script is started, the engine binds its symbols and immediately processes its root node, thus adding its outgoing edges to **waitingEdges**. The algorithm is an infinite loop that takes an event out of the input **eventQueue**, takes the matching edges out of the waiting set as well as their alternatives in case they have been queued by a command-executing scene or a join node, and processes the destination nodes of the matching edges. Processing (not shown) may queue new edges as well as changing situations, giving commands or starting subscripts as discussed above. Observe that this algorithm allows to easily capture ("checkpoint") and restore the state of the script engine before processing the event queue.

```
 1: var waitingEdges = set of edges;
 2: var eventQueue = FIFO list of events;
 3: while true do
 4:     wait UNTIL eventQueue NOT empty
 5:     for all evt IN eventQueue do
 6:         for all edge IN edgeList MATCHING evt do
 7:             remove edge FROM waitingEdges
 8:             remove alternatives FROM waitingEdges
 9:             process edge's destination node
10:         end for
11:         remove evt FROM eventQueue
12:     end for
13: end while
```

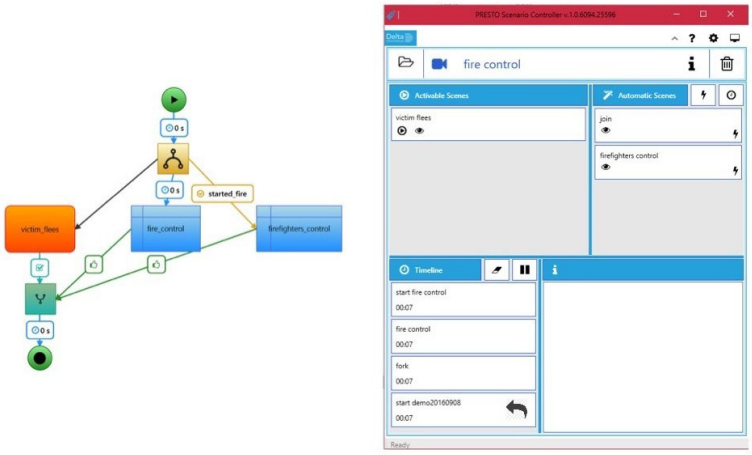
4.3 Implementation: editor, engine, controllers

The PRESTO Script editor, developed for Windows, is a GUI that supports the editing of scripts as well as of situation templates and includes browsers on PRESTO's metadata, including the ontology in use by the destination game for classification and qualities, the available agent roles and agent types of the NPCs, and so on. The editor integrates a cross-referencing facility that shows how scripts invoke each other, which situations they share, and other details useful for analyzing dependencies.

The PRESTO Script Engine implements the logic described above and offers APIs that can be called from anywhere to start scripts; this may include a player's UI or an agent, if desired. Further, it offers a network protocol that allows to start and terminate scripts, to be notified of the scenes that have been processed and of those that are waiting to be processed as soon as an appropriate event occurs, to know which choices are currently available and to select one, to be notified of available checkpoints and to restore state to one of them.

The network interface is used by two kind of controller programs, which can run on different devices from where the virtual environment is executing. Both controllers allow to start any number of scripts at any time and to decide which path to follow when a running script allows a choice, as well as checking the current state of the engine and restore it to a past checkpoint scene. The right side of Figure 1 is a snapshot of the Windows-based controller GUI for the end-user, designed to be easily used by a trainer. The snapshot has been taken just after traversing the fork point of the graph on the left: the Timeline box (bottom left) shows the scenes crossed so far, those marked as checkpoints contains a button that can be clicked to return back to that scene; the Automatic Scenes box (top right) shows which scenes will be crossed next as soon as specific events happen or timers expire; the Activable Scenes (top left) are waiting for the director to decide whether to reach them by clicking on the appropriate button. The information box (bottom right) shows information

Figure 1. Example of script for EMT training (left) and snapshot of the controller (right)



about a scene selected from the other boxes.

The second controller is meant for command-line use. It was designed mainly to automate the invocation of scripts, which is useful e.g. to execute test suites and to run PRESTO as an unsupervised serious game or even as a simulation without players when scripts do not contain interactive choices.

5 Comparisons

A full usability study would be required to check if PRESTO Script achieved its goal of being an environment for end-user development, namely for instructional designers building serious games to achieve specific training objectives. Some feedback was collected during a laboratory with some 30 Computer Science students, part of the Agent Oriented Software Engineering course at the University of Trento in 2016. Students were given a Unity 3D-based simulation game, controlled by PRESTO, and had to develop their own stories by means of PRESTO Script. While this laboratory experience, which involved a captive and technically well versed audience, was extremely useful to improve the language, it was not set up to be a proper study. At the moment, PRESTO Script is used only by Delta Informatica and its partners for course development and there is no opportunity for a large scale study. Missing that, to provide a proper perspective we briefly mention here a few languages and systems that have been of inspiration for PRESTO Script, then we pinpoint its unique features.

Commercial, built-in game languages. Unreal Engine Blueprints,[9] for the Unreal Engine, is based on graphs and supports events, variables, `if` switches, loops, and behavioral trees. Custom events with common type parameters can be defined, but they must be triggered from code and are not automatically detected. Of course, there is no layer of abstraction above the underlying engine, nor ontological classifications.

CryENGINE's Flow Graph[10] is also based on graphs but events are simple signals without parameters. It is XML-based and interpreted, differently from all other languages in this category that need to be compiled to be used within a game.

Unity uScript[11] and Unity PlayMaker[12] are two products for Unity 3D. The first is graph based (with events, `if` switches and variables); paths can be followed in parallel. The second adopts FSMs; parallels actions can be executed only within a single state.

Blender is a well known 3D modeling environment that supports scripting functionalities by means of BlenderLogic.[13] This is a game logic tool based on graphs with three types of nodes: actuators (to perform action in the VR), sensors (event listeners), and controllers (to evaluate logical expressions).

Visual programming for education. Two well known examples are Microsoft's Kodu[14] and MIT's Scratch.[15] They offer all classic imperative languages constructs, such as loops and `if` switches, and variables (in Scratch only). Declarative event-action rules allow to launch scripts, also in parallel, when prerequisites are verified. The focus of these languages is on simplifying the interaction and maximizing the enjoyment of programming for non-technical users rather than maximizing efficiency and flexibility.

Game research. DSVL [10] allows the fast development of electronic adventure text games. An adventure is described as a FSM, able to execute parallel actions e.g. to show messages on the screen.

Sketch'ndo [13] is a framework based on the Blender Game Engine for the creation of task-based serious games, with a player-centered story. A narrative editor allows a domain expert to write tasks as trees of sequential and parallel action blocks. The language supports also pre-conditions and consequent actions, represented as FSMs.

CANVAS [9] empowers non-trained authors to create their own animated stories, without having to write all single details. Each story consists in a sequence of blocks, where each block can contain a set of actions that run in parallel. Pre-conditions and post-conditions can be specified, allowing partial stories that can be completed by an automatic planner.

Simulation. Simulink Stateflow,[16] based on MathWorks, is a scripting language that can generate C code, starting from the definition of a hierarchical FSM.

[9] https://docs.unrealengine.com/latest/INT/Engine/Blueprints/; this URL and all the following ones have been checked during December 2016
[10] http://docs.cryengine.com/display/CEMANUAL/Flow+Graph
[11] http://www.uscript.net/home/
[12] http://www.hutonggames.com/
[13] https://www.blender.org/manual/game_engine/logic/introduction.html
[14] http://www.kodugamelab.com/
[15] https://scratch.mit.edu/
[16] https://it.mathworks.com/products/stateflow/

LabView,[17] by National Instruments, supports data flow charts to describe logical circuits where a source signal flows through wires and blocks to reach an output.

General purpose. A number of workflow languages, with visual and textual syntax, represent procedures involving multiple actors, even if only a few are executable. The closest to PRESTO Script's approach is BPEL,[18] an XML-based Web Service orchestration language standardized by OASIS.

Discussion. For lack of space, unfortunately it is not possible to provide a detailed comparison with the languages and environments mentioned above nor a full discussion. Here, we highlight some of the specificities of PRESTO Script that stand out:

- PRESTO Script is not a visual language, even if its editor adopts a graphical representation of the graph of scenes. The prerequisite section and many details of scenes are edited via plain forms. While this approach may seems incomplete in terms of usability, having a XML syntax opens the way to alternative editors, planners and other code generation techniques;
- PRESTO Script is fully interpreted. Scripts can be edited while the game is running. The engine always loads the latest version from the file system whenever a script is invoked. Note that this may enable the introduction of script generators that act on-the-fly;
- game evolution is determined by the interplay of multiple scripts, especially as asynchronous subscripts. The language semantics enables the analysis of sets of scripts, as well as checkpointing and restoring the state of all running scripts at specific times;
- the language supports a specific user role, a game master (typically, a trainer), allowing him to control the script execution;
- scripting is game- and model- independent, thanks to the use of semantically rich meta-data (i.e., agent roles and types, ontologically defined qualities and classifications of entities and locations, situations) to access the underlying engine. To put it in another way, any game-specific capability is automatically supported by PRESTO Script as long as it is captured by meta-data;
- among the event types supported in PRESTO Script, two of them concern situations, which can be defined with a powerful language that supports parameters and quantifiers;
- PRESTO Script is suitable to represent coordination among NPCs, while individual behaviors are delegated to the latter's own models. PRESTO supports any NPC modeling technique as long as it has an appropriate meta-data representation. If DICE is used, then the developer can access to two other interpreted languages for NPC-specific modeling (see Sec. 3);
- scenes do not have a duration, so by definition they are not interruptible. An important consequence is that goals given to NPCs by a scene are not affected when an event causes the next scene to run. Thus, evolution of a story requiring changes of behavior must either be managed by the script itself or by NPC models. Concerning the latter, worth noting is that DICE supports two-level decision-making intentions (reactive and planned) whose goals can be changed instantly, so context-sensitive adaptable behaviors can easily be developed.

A few future research directions for PRESTO Script clearly emerge:

- improving and completing its graphical representation, including full integration with DICE Parts;
- supporting the interaction between scripts and players / trainees, in addition to the trainer;
- supporting situations representing the players' state, e.g. reflecting their current performance or emotional state.

[17] http://www.ni.com/labview/

[18] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

6 Conclusion

PRESTO covered a lot of ground in the area between training conception and serious game development. PRESTO Script is an environment that includes a language, a visual editor, an engine and an interactive controller that have been conceived for instructional designers and trainers. Some of its characteristics make it unique in the current landscape of scripting languages. Further research is required both to verify its usability outside the current captive user base and to extend it, especially to take in account the player's state.

References

[1] Michael E. Bratman, *Intention, Plans, and Practical Reason*, Harvard University Press, November 1987.

[2] P Busetta, M Fruet, P Consolati, M Dragoni, and C Ghidini, 'Developing an ontology for autonomous entities in a virtual reality: the PRESTO experience', in *Proceedings of MESAS 2015 workshop*, Prague (CZ), (2015). Springer LNCS.

[3] P Busetta, C Ghidini, M Pedrotti, AD Angeli, and Z Menestrina, 'Briefing virtual actors: a first report on the presto project', in *Proceedings of the AI and Games Symposium at AISB*, London, (2014).

[4] Paolo Busetta, Nicholas Howden, Ralph Rönnquist, and Andrew Hodgson, 'Structuring BDI Agents In Functional Clusters', in *Proceedings of the Workshop on Agent Theories Architectures and Languages (ATAL-99)*, volume 1757 of *Lecture Notes in Artificial Intelligence*, Orlando, Florida, (15-17th July 1999). Springer Verlag.

[5] Paolo Calanca and Paolo Busetta, 'Cognitive Navigation in PRESTO', in *AI&Games workshop @ AISB 2015*, Canterbury (UK), (2015).

[6] Daniel D. Corkill, 'Blackboard systems', (1991).

[7] M Dragoni, C Ghidini, P Busetta, M Fruet, and M Pedrotti, 'Using Ontologies For Modeling Virtual Reality Scenarios', in *Proceedings of ESWC 2015*. Springer, (2015).

[8] Rick Evertsz, Frank E. Ritter, Paolo Busetta, Matteo Pedrotti, and Jennifer L. Bittner, 'CoJACK – Achieving Principled Behaviour Variation in a Moderated Cognitive Architecture', in *Proceedings of the 17th conference on behavior representation in modeling and simulation*, pp. 80–89, (2008).

[9] Mubbasir Kapadia, Seth Frey, Alexander Shoulson, Robert W. Sumner, and Markus Gross, 'Canvas: Computer-assisted narrative animation synthesis', in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '16, pp. 199–209, Aire-la-Ville, Switzerland, Switzerland, (2016). Eurographics Association.

[10] Eugenio J. Marchiori, Ángel del Blanco, Javier Torrente, Iván Martinez-Ortiz, and Baltasar Fernández-Manjón, 'A visual language for the creation of narrative educational games', *Journal of Visual Languages & Computing*, **22**(6), 443 – 452, (2011).

[11] Paolo Busetta Marco Robol, Paolo Calanca, 'Applying bdi to serious games: The presto experience', Technical Report DISI-16-011, University of Trento, http://hdl.handle.net/11572/154697, (2016).

[12] Z Menestrina, A De Angeli, A De Angeli, and P Busetta, 'APE: End User Development for Emergency Management Training', *6th International Conference on Games and Virtual Worlds for Serious Applications VS-GAMES 2014*, (2014).

[13] Sergio Moya, Dani Tost, Sergi Grau, Ariel von Barnekow, and Eloy Felix, 'Sketch'ndo: A framework for the creation of task-based serious games', *Journal of Visual Languages & Computing*, **34–35**, 1 – 10, (2016).

[14] AS Rao and MP Georgeff, 'BDI agents: From theory to practice.', in *Proceedings of the 1st international conference on multi-agents-systems (ICMAS)*, pp. 312–319, Menlo (CA), (1995). AAAI Press.

[15] FE Ritter, JL Bittner, SE Kase, and R Evertsz, 'CoJACK: A high-level cognitive architecture with demonstrations of moderators, variability, and implications for situation awareness', *Biologically Inspired Cognitive Architectures*, **1**, 2–13, (2012).

[16] Marco Robol, Paolo Giorgini, and Paolo Busetta, 'Applying social norms to implicit negotiation among Non-Player Characters in serious games', in *Proceedings of WOA 2016*, Catania, (2016).