# Applying BDI To Serious Games: The PRESTO Experience

Paolo Busetta[2], Paolo Calanca[2], Marco Robol[3]
*[1] Delta Informatica SpA, paolo.busetta@deltainformatica.eu*
*[2] Delta Informatica SpA, paolo.calanca@deltainformatica.eu*
*[3] Universita' di Trento, marco.robol@unitn.it*

December 2016

Technical Report # DISI-16-011
Version 2.0

# Applying BDI To Serious Games: The PRESTO Experience

Paolo Busetta, Paolo Calanca, Marco Robol

**Abstract**—This article summarizes the main results of PRESTO, a 4-year industrial research project ending in 2016. Its main objective was the development of an environment for the authoring and control of training sessions in 3D-based serious games, especially in the domain of emergency management. PRESTO adopts artificial intelligence and reusable components (models of behavior of NPCs and game-level scripts) and offers end-user tools for their development, in addition to programming frameworks and APIs. The starting point was the authors' experiences with the BDI (Belief, Desire, Intention) agent architecture and its cognitive extensions. Given the relative immaturity of the selected game environments (Unity and XVR) from a behavioral AI perspective, a significant number of challenges concerning semantics, perception and control had to be tackled. Results presented here include game-agnostic semantization, composition of tactical agents from reusable behavioral models, agent-level scripting for game-specific behaviors, game-level scripting to represent agent and game strategies, and graphical development environments directed at game and domain specialists rather than software engineers. An analysis methodology and a few pilot studies are introduced. The PRESTO suite, used for the development of commercial training services, is currently available only on request, for both commercial and research uses.

**Index Terms**—BDI, Autonomous Agents, Artificial Intelligence, Virtual Reality, Simulation, Serious Games, Behavioral Models.

## 1 Introduction

PRESTO (Plausible Representation of Emergency Scenarios for Training Operations) was an industrial R&D project, led by Delta Informatica SpA, based in Trento (Italy), and involving various research centers specialized in knowledge capture, representation and cognitive modelling, including the University of Trento. The idea at the basis of PRESTO was to create an all-round development environment for NPC's behaviours, starting from the authors' experiences in multi-agent systems for decision-support and for developing "intelligent opponents", mostly for small-scale military 3D serious games. With NPC's behaviours, we refer to decision-making at cognitive level, specifically focusing on tactical reasoning concerning how to achieve goals, rather than either instantaneous, instinctive behavior or long-term planning. The main technical objectives of the project were: enabling NPC's model reusability; creating modeling environments designed for trainers rather than software engineers; supporting the intervention of a trainer on the PRESTO-controlled game to influence decisions and the unfolding of events. The main business objective was to ease adoption of serious games in non-military emergency training and other serious and entertainment domains requiring NPCs to show complex, believable behaviors adapted to specific needs (i.e. ad-hoc training scenarios for a small group of trainees) with very limited budget and time with respect to commercial videogames made for mass markets.

Progress towards these ambitious goals was hampered by a number of unanticipated obstacles and research challenges greater than expected. Concerning specifically NPC decision-making, the encountered issues can be classified in three main areas:

- Immaturity of the chosen game environments (Unity[1] and XVR[2]), if not of game engines in general, concerning common behaviours, most importantly navigation. Many more resources than originally planned had to be spent on the latter and on other relatively low-level behaviors, e.g. avatars' posture and low-level actions such as pushing and pulling, also to influence them with cognitive states. Some interesting results have been reached (see e.g. [1]), which are not going to be discussed here;
- Immaturity, or perhaps excessive genericity, of the chosen BDI (Belief Desire Intention [2]) systems, JACK[3] and CoJACK[4] [3] [4] [5], with respect to the requirements of an agent architecture for PRESTO. To this end, meta-data and meta-level reasoning facilities had to be added, semantic services were integrated, and various technological issues with multi-language developments (Java and C#) had to be dealt with, as briefly mentioned later;
- Immaturity of methodologies for modeling behaviors and training scenarios in BDI terms. While agent-oriented analysis methodologies are available

- *P. Busetta and P. Calanca work for Delta Informatica Spa, Trento, Italy. E-mail Busetta: paolo.busetta@deltainformatica.eu; E-mail Calanca: paolo.calanca@deltainformatica.eu*
- *M. Robol is at the University of Trento, Italy. E-mail: marco.robol@unitn.it*

1. Unity, unity3d.com [Accessed 19 Sep 2016]
2. XVR, www.xvrsim.com [Accessed 19 Sep 2016]
3. AOS JACK, www.aosgrp.com/products/jack [Accessed 19 Sep 2016]
4. AOS CoJACK, www.aosgrp.com/products/cojack [Accessed 19 Sep 2016]

(see e.g. Tropos[5] and Prometheus[6]), they are rarely adopted in industrial practice; further, going beyond a procedural or simple rule-based representation of expected behaviors to methodically capture e.g. emotion-driven or skill-driven differences among people is a challenge. Some initial work undertaken in PRESTO is discussed later.

Notwithstanding the difficulties mentioned above, PRESTO has produced a significant software base, embodied in prototypes of tools at various stages of readiness, experimented in three pilots (a hospital risk-management course, delivered in multiple editions to hundredths of trainees, and two University laboratories on game development that produced more than 40 student projects), and reported in a few research publications. While one can argue about how far its objectives are from being achieved, in our opinion the current results show that they are reachable. PRESTO' main contributions to the state-of-the-art can be summarized as follows:

- Semantics: entities and locations in the VR are not simply classified but can be dynamically tagged to enable arbitrary reasoning, including on actions and goals, by observed entities. This is exploited to dynamically detect situations, representing a further abstraction that allows the writing of truly adaptable, reusable models;
- Modular and dynamically adaptable agent architecture: thanks to meta-data and to its agent framework, a NPC-controlling agent in PRESTO can be built by composing an arbitrary number of models, potentially achieving the same goals but in different ways, and dynamically selecting a profile that determines the NPC's current behaviours. In turn, this profiling supports the creation of meta-models of coordination, emotions and other physiological or social factors without having to hardcode them in the logic of behaviours;
- End-user programming environments: meta-data and goal-oriented programming allow the introduction of high-level languages that allow a domain or game expert to write custom agent plans and game scripts that submit sequences of goals and control their execution according to events unfolding in the game. To make this language suitable to a non-programmer, generality is traded-off for expressivity and graphical editors are provided.

### 1.1 Article structure

This report, at its second version (the previous one was extended with more details on semantics and scripting) summarizes PRESTO's main outcomes and discusses its pilot studies that are the basis for Delta Informatica's virtual-reality based training services. Next section gives an overview of PRESTO, its architecture and main components. Sec. 3 discusses semantic representations available in PRESTO, while Sec. 4 provides more details on a specific one, called "situations". Sec. 5, 6 and 7 focus on PRESTO's behavior engine, called DICE, with a closer look at its features for reusability and meta-modelling. Sec. 8 presents PRESTO's game-level scripting environment. The pilot studies are briefly presented in Sec. 9. Sec. 10 introduces a methodology for behavior and script development.

December, 2016

## 2 Overview of PRESTO

PRESTO started in 2013 and concluded in November 2016. PRESTO was led by Delta Informatica and was subsidized by a grant of the local government: the Provincia Autonoma di Trento (PAT), Italy. It involved three local research centers: FBK, DISI of University of Trento, and CIMEC. Its objective was the creation of a development environment for reusable, game- and platform-independent NPC behaviours, enabling adaptation by the end-user and eventually a marketplace of models. The anticipated modality of use of PRESTO was to enhance virtual environments with "intelligent" NPCs to support serious games development. To cover the widest possible range of development requirements, various game modalities had been taken into consideration, e.g. multi-player training games with a supervisor (a trainer or game master) determining the unfolding of events, unsupervised single-player games, simulations with no players involved in first person but rather acting on configurations and game plot. The main domain of application, as in the pilot project described later, was emergency management training (EMT) directed in particular to operatives and coordinators of public facilities (e.g. hospitals).

At the end of the project, in addition to software architectures and a methodology for capturing stories from domain experts, PRESTO has produced:

- A modular and extendable framework for the integration of NPC-controlling agents in generic videogames;
- a tool suite, described in this paper, for the production and run-time control of models of behaviours of NPCs and game scripts, including languages and editors directed to domain experts rather than software programmers;
- a simulation game called PUG (PRESTO Unity Game), built in-house with Unity 3D, and a PRESTO plug-in for a commercial 3D EMT product, XVR by XVR Simulation (formerly E-semble).

Figure 1 summarizes PRESTO and its expected usage context. The boxes within the blue area are components created by the project. They include:

- a methodological tool suite directed to a game writer (the "training strategist" in the picture);
- a set of graphical editors for scripting the behavior of individual NPCs as well as the overall game meant to be used both by the game writer and by the game master (the "instructor") before running a session;
- a (third-party extensible) library of pre-built, composable BDI models;
- two major engines (one for the agents, called DICE, and one for the game, indicated as "Director Assistant") ;

5. Tropos, www.troposproject.org [Accessed 19 Sep 2016]
6. Prometheus, sites.google.com/site/rmitagents [Accessed 19 Sep 2016]

Figure 1. PRESTO overview

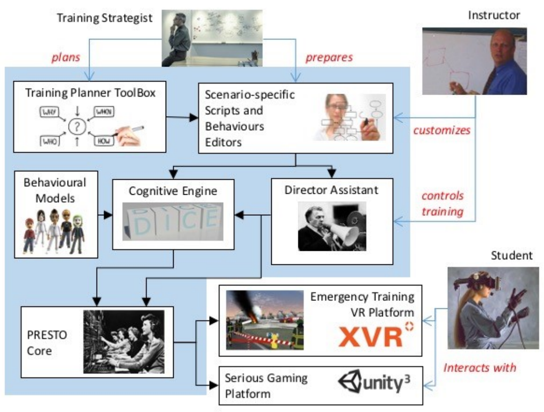
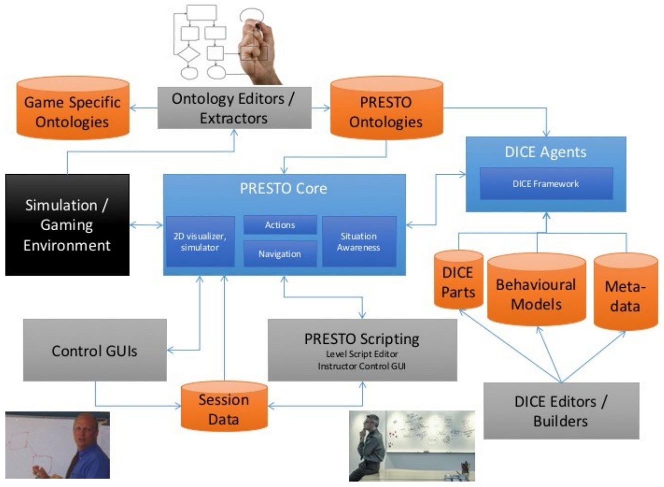


Figure 2. Component view



- a set of editors and run-time APIs and facilities (the PRESTO core) that allows agents and scripts to interact with the virtual environment.

Notably, interaction with human players is outside of the scope of PRESTO and left to the integrated games (e.g. XVR or a generic Unity game), even if some initial work has been done and is an area for future developments.

PUG (built as part of PRESTO itself) and XVR have been used to test the tool suite in the pilot projects, as discussed later. Further, PUG has been used as a base for a number of experimental systems, of which two [6] [7] were student projects exploring novel UIs directed at casual players not familiar with videogames.

The team working on PRESTO has included: within Delta Informatica, an average of one researcher, one senior engineer, 2 developers for the entire lifetime of the project; at least one research per each involved research centers (three overall on two-year contracts each); further, a dozen students of Computer Science and Engineering at various course levels and with different purposes and modalities (4 to 8 weeks of professional stages, bachelor and Master thesis lasting from 3 to 6 months, short studies of various length for PhD-level research).

## 2.1 Architecture and baseline technologies

Figure 2 summarizes the major functional components of PRESTO and how they are related. Blue boxes represent AI run-time components, linked with the serious game; orange cylinders, data structures; grey boxes, UIs and utilities used both on- and off-line; finally, the black box is the graphics or the simulation engine integrated with PRESTO.

PRESTO is based on many technologies, proprietary and open-source. To mention the most important: the bulk of the core and many utilities are C#, while agents are mostly a mix of Java and AOS JACK. IKVM (www.ikvm.net) is adopted to convert Javas bytecode in its C# equivalent, enabling both languages to run in a single executable process. Network connections among distributed components (e.g. control GUIs to PRESTO core) adopt mostly ad-hoc protocols partially built with open-source frameworks on top of TCP/IP streams. PUG and part of the XVR plug-in have required the usage of Unity 3D; non-3D UIs, including editors, are mostly in C# with Microsofts WPF framework. Ontologies are in OWL, edited with Protege (protege.stanford.edu) and accessed with Sesame (currently called Eclipse RDF4J); also, many internally built tools convert ontologies in various forms of constants and configurations for Java and C#. XML and related Java and C# frameworks are extensively used for data storing. PRESTO has been tested and used on Windows only, even if potentially its core components could run on any Mono- and Unity 3D-supporting operating systems.

This very complex development environment has, unavoidably, a number of drawbacks. To mention a few: IKVM enables the support of JACK and Sesame in C# programs but creates a number of hassles concerning in particular debugging and version compatibilities when integrating with games and game platforms. This includes a quite inefficient develop / compile / debug cycle, partially worked around by the implementation of interpreters (DICE Parts and PRESTO Script, discussed later). On a design level, networking within the core components has to be carefully reexamined in particular to support the distribution of agents, since even minimal latency is unacceptable given the strict real-time requirements of perception and gaming.

## 2.2 Main software artifacts

A few of the software elements developed within PRESTO are worth mentioning on their own:

- the DICE agent framework, including its DICE Parts interpreted language, for the implementation of tactical behavior of NPCs (see Sections 5 and 6);
- some reusable behavioral models and underlying semantics of objects and actions, including cognitive path planning [1] and implicit coordination (Sec. 7);
- PRESTO Script, which includes an engine, a controller UI and an editor (Sec. 8) and is complemented by the situation engine (Sec. 4), for overall game control;
- semantic representation formats and structures as well as a number of complementary meta-data

schemas that are independent of the background technology and that may reusable in the design of other game AI middleware (see Section 3 and 4 in particular, but meta-data is pervasive in PRESTO).

All of the above relies on a large software base, represented by the PRESTO Core box in Figure 2 above, that supports integration of game engines, abstraction of their entities and extraction of semantically annotated streams of perceptions and other data. All components are largely configurable; in particular, DICE supports configuration down to each individual agent.

Not all elements mentioned above have reached the same level of maturity at the end of the project. For instance, the DICE Part language and its editor are still subject to revisions, while the PRESTO Script environment is ready to the point of being considered for non-gaming uses.

## 3 Virtual environment semantics

To agents controlling NPCs' behaviours 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 theatre" 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 [8], [9], 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 an association table that provides the classification of a perceived object together with perception data to the agents. [8] 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.

Qualities can be "functional", that is, take a single value, or "relational", that is, take multiple values. Functional qualities often represent game-specific information, synchronized in both directions (game to PRESTO and viceversa) by the PRESTO integration layer. They include states (such as posture of avatars, open/close position of doors) and attributes enabling actions (such as "crossable" for doors and any object that needs to be opened during navigation). Relational qualities are mostly used to represent relationships between entities and locations, for example spatial ("isInside" or "hasInside"), and coordination information (such as "engaging" entities [10], [11]). 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. 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 [12]. 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).

In order to enable syntax checking and automatic monitoring, the situation engine and the PRESTO graphical editors use *situation templates* defined in configuration files, described in detail below (Sec. 4). In short, a template specifies the name of a situation as well as the names and types of its parameters. Further, the template for a situation to be automatically monitored contains a boolean expression and, optionally, one or more symbols that are bound to entities and locations according to filtering patterns before the expression is evaluated by the situation engine. As described later, this expression may contain quantifiers and checks on the qualities of the objects bound to symbols and parameters; for instance, the template for "Fire-in-Room" may have a "$room" parameter which is a location's name, a symbol ''$fire" to be bound to entities classified as fires, a symbol "$victim" to be bound to avatars, and an expression meaning "at least one entity in $fire has the quality isInside $room and at least one $victim isPerforming coughing".

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 situtations, 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.

## 4 Situation Recognition

As mentioned earlier, situations are one of the three forms of semantic representation provided by PRESTO. This section focuses on their declaration language and on the engine that maintains their truth value.

### 4.1 Templates, Symbols and Expressions

Situation templates, edited with a specialized editor, are used to declare classes of situations to be used by agents and scripts. A template specifies the name of a situation and its formal parameters, i.e. the parameters' names and types that have to be assigned at run-time. For example, a template for the situation "fire_inside_room" may have a "room" parameter; a script may declare that the situation "fire_inside_room (room = reception)" is true.

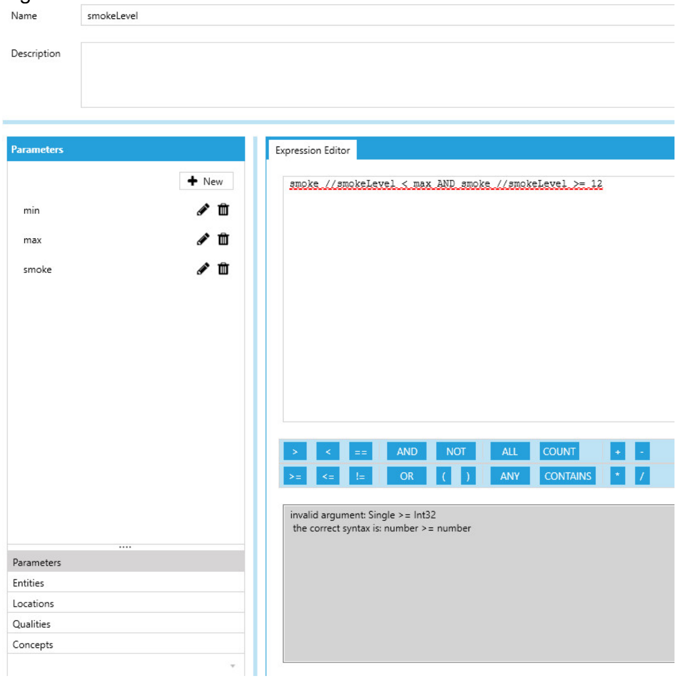
To enable automatic monitoring, a situation template must contain an expression defined with the *situation language* described below. In short, this supports boolean expressions that contain template-specific symbols and parameters names, ontological classes, qualities. A symbol is described within the template by specifying filtering criteria used to match entities or locations at run-time, similarly to script symbols described later. Filtering criteria may include an entity's or location's name, its ontological class, its associated agent role and type (described with DICE later). At run-time, before evaluating the expression of a situation, its symbols are bound to the set of entities or locations that satisfy their filters; for instance, a "firefigthing_team" symbol could specify that the entities to be bound must be classified as "Emergency Personnel" and must support the role of "Firefigther".

Figure 3 shows a screenshot of the situation editor. The left side shows a definition of parameters and symbols while the right side contains the expression editor, where the user can write an expression by typing or dragging symbols from the left panel and combining them with the operators. The expression is parsed interactively to check for syntax errors (e.g. undefined symbols, parenthesis mismatches, wrong types or wrong number of operator arguments), which are shown in the panel beneath.

### 4.2 Situation Language

The situation language allows the writing of boolean expressions concerning the virtual world's state, in terms of items (entities or locations), their properties (functional qualities) and relations (relational qualities). These boolean expression are composed by predicates, which allow the comparison

Figure 3. Situation defined in the PRESTO Editor



Name: smokeLevel

min
max
smoke

```
smoke //smokeLevel < max AND smoke //smokeLevel >= 12
```

```
invalid argument: Single >= Int32
 the correct syntax is: number >= number
```

between items' properties and base values and relations among items. 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 and locations symbols. Counting quantifiers can also be used to count the items that have a specific relation with a specified item.

Formally, the situation language is defined by the pair ⟨S, G⟩, that is, a context-free grammar G and a set of symbols S composed by:

- Binary predicates: **and**, **or**, **not**
- Comparison predicates (COMP): $==, !=, <, <=, >, >=$
- Binary operation functions (OP): +, -, *, /
- Parenthesis: ( , )
- Constants (CONST): floating point numbers ending with 'f', integer numbers, boolean values (**true**/**false**), strings enclosed by ' '
- An operator to access qualities: //
- A predicate to check inclusion: **contains**
- Quantifiers: **all**, **any**, **count**
- Functional qualities (FQ), defined in the ontology in use
- Relational qualities (RQ), defined in the ontology in use
- Ontological classes (CLASS), defined in the ontology in use
- Entities symbols (ENTITIES), defined within the situation; they refers to one or more entities according to specified filters
- Locations symbols (LOCATIONS), defined within the situation; they refers to one or more locations
- Parameters (PARAM), defined within the situation and identified with a name. Supported parameter types are: base values (VALUE_PARAM), quali-

ties (FQ_PARAM, RQ_PARAM), entities (E_PARAM) and locations (L_PARAM).

The grammar G is defined by the followings production rules:

predicate: $P \Rightarrow (P) |$ **true** | **false**

$P \Rightarrow$ **not** $P | P$ **and** $P | P$ **or** $P$

$P \Rightarrow V$ COMP $V |$VS **contains** $(V)$

value set: VS $\Rightarrow$ ITEM // RQ | ITEM // RQ_PARAM

value: $V \Rightarrow$ CONST | VALUE_PARAM $| V$ OP $V$

$V \Rightarrow$ ITEM // FQ | ITEM // FQ_PARAM

$V \Rightarrow$ **count** (VS) | **count** (ITEM)

items: ITEM $\Rightarrow$ **all** I | **any** I | I

I $\Rightarrow$ E_PARAM | L_PARAM | ENTITIES | LOCATIONS

The quality operator (//) reads the quality specified as second parameter that is associated to its first parameter and returns a single value in case of a functional quality or a set of values in case of a relational quality. In the second case, **contains** can be applied to check if that set contains a specific item. For example, to check the relation (fire, isInside, reception):

fire //isInside **contains** (reception)

To compare functional qualities and values:

fire //intensity $> 3.5f$

The following expression describes the fact that the reception has at least one item inside:

**count** ( reception //hasInside ) $>= 1$

The number of items bound to an entity symbol can be compared to a value:

**count** (firefighter) $> 0$

Universal or existential quantifier can be applied to entities and locations symbols; for example, the following expression checks the fact that all items bound to the firefighter symbol, whose filtering criteria may specify that they belong to the "Firefighter" class, are inside the reception:

**all** firefighter //isInside **contains** (reception)

The universal and existential quantifiers are resolved after that their enclosed sub-expression (which may include quality operators, OP functions, COMP and **contains** predicates) is evaluated on all items bound to their first parameter.

### 4.3 Situation Engine

The *situation engine* is the component that, at run-time, monitors the situations tuples, updates their truth values and notifies subscribers (which may be agents and the script engine described later) of changes to these values. The subscribers request the engine to evaluate a tuple by providing the situation's name and the values of the parameters; optionally, subscribers may also manually provide the truth values. If a situation expression is found within the template of a situation, the truth value for a requested tuple is automatically updated. To this end, the situation engine uses four components: (i) a symbol resolver, (ii) a parser, (iii) an interpreter, (iv) a situation monitor.

(i) Initially, the *symbol resolver* looks for entities and locations that match with the symbol descriptors and stores them.

(ii) The expression is then *parsed*. The parser first checks for syntax errors and, if there are none, it builds an expression tree where each node represents on operator and its children its parameters, which in turn can be other operators.

(iii) The expression tree is then used by the interpreter to evaluate the situation. The *interpreter* and the expression tree are implemented using the efficient .NET Dynamic Language run-time facilities which allows to define the tree operators and parameters and to dynamically generate the corresponding code. This is stored and then used each time the situation needs to be evaluated.

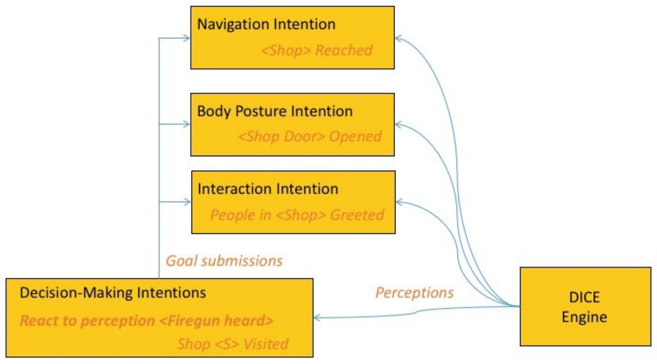
(iv) The *situation monitor* updates the truth values of a situation by evaluating its expression when necessary, that is, when a quality used in the expression changes for one of the items in the situation's symbols and parameters or when an item that matches a symbol descriptor is added or removed from the virtual reality (both these types of events are notified by the PRESTO core).

## 5 The DICE agent framework

PRESTO relies on autonomous agents to bring AI to video games, in particular to control NPCs. DICE [8] [13] is a BDI agent framework built on top of JACK for controlling NPCs of a game encapsulated by PRESTO, which offers services such as situation awareness and game-independent action execution. Typically, a DICE agent controls one NPC. DICE supports the needs of PRESTO's scripting: 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 (e.g. by exploiting the "default" and "idle" goals automatically submitted by DICE to an agent's so-called "prevailing" role) as well as of scripts that represent detailed procedures at the tactical level and even manipulate graphical details.

DICE's core implements an update / decide / act loop, reminding of the classic OODA (Observe / Orient / Decide / Act) loop [14], driven by the perception flow generated by PRESTO's situation awareness; this flow is used to refresh short-term memory and advance the agent's state. DICE's core loop is designed to guarantee critical sections between

Figure 4. A (simplified) snapshot of a DICE agent



updates (corresponding to the "cognitive steps" of CoJACK), to trigger any model-specific reaction at well-known times and to schedule decision-making intentions in well-defined, non-preemptive order. 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. The control of independent body functions (e.g., eye gaze, head movements, speech, hand gestures, locomotion) is performed by intentions called "executors" running concurrently with the decision-making ones, with the latter delegating goals to the former and coordinating their execution. 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. Figure 4 illustrates an example of an agent pursuing a high level goal ("shop *s* visited") whose current plan, in turn, has submitted the "people greeted", "door opened", and "place reached" goals to the relevant executors; the perception of the noise of something dangerous has started a reaction that will pause the main goal until over, most likely changing the goals of the executors in the meantime.

As discussed later, DICE provides various types of meta-data and introspection facilities on top of JACK that enable model composition, plan interpretation, emotional influences, 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 concepts, 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" concept may be used to annotate very different plans or goals applied by different agent models when performing any task that requires using a device of type D).

## 6 Agent behavioral profile. Part scripting

A DICE agent *type* is constructed as a composition of *roles* and *behavioral models*; a role defines a set of goals, while a behavioral model implements the tactics required to achieve the goals of a specific role. In turn, behavioral models may depend on other roles, so starting from one or more high level roles for an agent (one of which is dynamically selected as prevailing) a graph of roles and models is derived, partly automatically (because of dependencies) and partly by the modeler's choice. As mentioned above, DICE automatically submits a goal, called "default", to the prevailing role when an agent is created and the "idle" goal when there is no (or no longer a) high-level goal to be satisfied. This allows the modeler to easily build fully autonomous NPCs that know what to do by themselves, while not managing "default" and "idle" will simply leave the NPC waiting for goals.

An agent can be equipped with multiple behavioral models implementing the same role; the set of active models (called the current behavioral profile) is defined dynamically, according either to internal rules or to external API calls. This mechanism greatly simplifies the approach to handling cognitive states, which in CoJACK mainly acts at plan selection level. It allows to equip a single agent with the ability to pursue its goals with different skills levels or performance characteristics according to cognitive parameters changing over time. So-called moderators (which are simply percentages) are exploited to represent emotions or physiological factors, such as fear and fatigue levels, and can dynamically trigger the selection of a specific behavioral model between the available ones. For instance, a role "hospital nurse" that accepts goals such as "take care of patient P" may be implemented by a "highly skilled, calm" model, an "intermediate skilled, tense" model, and a "poorly skilled, careless" model; a generic "nurse" agent type will select which one to activate according to the desired behavior of a specific NPCs, which in turn may depend on a combination of the fear and fatigue moderators.

A few experiments, whose results are yet to be published at the time of writing, have been performed to identify a representation and a computational model to provide DICE agents with an emotional state. Emotions have been represented in a PAD (Pleasure, Arousal, Dominance) space, as proposed by Becker-Asano et al in (WASABI, 2004) [15]. A computational model has been developed to describe evolution of emotions over time, according to PAD values associated to events occurring during a game session (see the methodology section later). The resulting emotion have been exploited to dynamically adjust values of moderators and other cognitive parameters and, consequently, to induce behavioral profile change. Given the immaturity of this approach, in current practice the behavioral profile for a specific NPC is chosen according to defaults and explicitly changed only via scripting (e.g., in the example above, an NPC configured as a highly skilled nurse at the beginning may be forced to a lower quality behavioral profile by a game script when this generates a stressing situation).

A few other aspects of DICE have been inspired from CoJACK, most importantly memory management. Individual agents are configured with their own background knowledge, which boils down to a-priori knowledge about entities, locations, and other environmental data defined by means of semantic queries. Short term memory is fed from perceptions and is periodically cleaned up by a behavioral model for cognitive management according to a number of parameters. Consequently, agents of the same type may behave differently because they know different things, possibly because they forgot them, or because they have a different behavioral profile.

Goals for the agent are dispatched by DICE to the appropriate behavioral model according to the current behavioral profile. Goals may be generated internally to the agent by a running plan or arrive from the outside world. Internal goals are typically sub-goals and thus pushed on the same decision-making or execution intention stack of the submitting plan. By contrast, all externally generated goals (e.g. a command pushed by a script) as well as properly marked internal goals (e.g. a reaction to a new situation, a new navigation destination determined by a decision-making intention) are handled as root goals for their destination intention stack, causing the controlled shutdown of the previous root goal in the same stack if there was one. This is implemented via the maintenance condition mechanism of JACK.

DICE supports two interpreted languages aimed at end-user development for the rapid creation of ad-hoc procedures rather than for complex, reusable modelling, 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 durations 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. This language is interpreted by a module external to DICE but pre-built in all agents, composed by a runtime interpreter behavioral model and role defining an execute goal. Game-level scripts can submit the execute goal to the agent with the text to be interpreted in input.

The second language is called DICE Part, where "part" is a term taken from theatre 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 [16] (Figure 5), implements a complete behavioral model; in its current version, it must contain a procedure (called a DICE Part, indeed) for each of the goals supported by its role and optionally reactions to specific events, including the detection of situations. A DICE Part is a sequence of steps and of a few conditional statement types (IF and WHILE). Each step can submit at most one decision-making subgoal and at most one root goal per each executor. Conditions are themselves based on events, including perceptions of specific types of entities, changes in internal state (represented e.g. by moderators), changes in situations, timers. Further, a FOR EACH statement allows cycling over entities known to the agent filtered according to semantic criteria

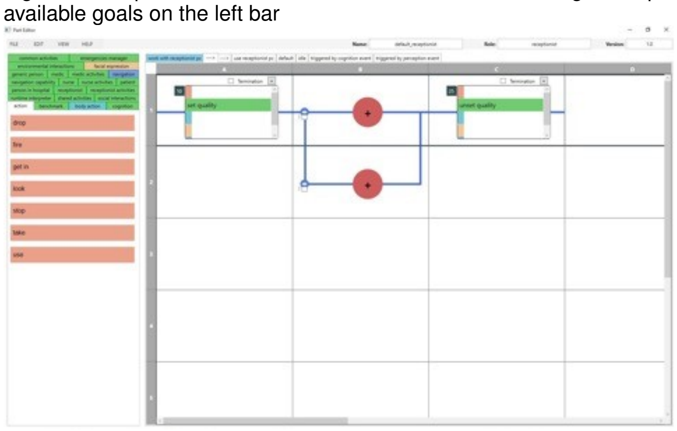


Figure 5. Snapshot of the DICE Part editor. Cells are multi-goal steps; available goals on the left bar

(in particular, their ontological classification and storage in short-term memory rather than background knowledge). No user-declared variable is supported; only parameters of the triggering event, inputs / outputs of invoked goals, and iterators of loops are available. This syntax allows writing procedures such as the following pseudo-code (SIT stands for "situation", described in Sec. 3):

```
// React to an evacuation situation
// (most likely, game script-controlled)
WHEN SIT <EVACUATION_IN_PROGRESS> TRUE:
    // Do what follows while alarm on
  WHILE SIT <FIRE_ALARM_ON> TRUE:
      // For all those you see at this time ...
    FOREACH PERCEIVED <PERSON>:
        // ... achieve these goals in parallel:
      BEGIN STEP
         // animate head as if talking
        STARTED CHAT WITH <CURRENT PERSON>
         // ... while moving the person out
        EVACUATED <CURRENT PERSON>
              TO <FIRE_SAFE_AREA>
      END STEP
    END FOREACH
  END WHILE
END WHILE
```

In terms of expressiveness, DICE Part is very powerful and compact with respect to JACK and thus often adopted by developers in spite of missing common, programmer-expected features such as simple Boolean expressions and assignments to local variables. A revision and possibly an evolution into two different dialects (one modeler-oriented, the other end-user / customization oriented) is likely as experience is collected. Worth noting is that DICE Parts could be easily generated by a planner or a code synthesizer, thus opening the way to the introduction of strategical AI, learning-by-example techniques, and so on, in addition to alternative editors to the default one.

## 7 Implicit coordination. "Agentification" of the human player

The goal-oriented nature (based on BDI) and the facilities provided by DICE have allowed the exploration of a novel technique for coordination within the environment, inspired by analogy with what humans commonly do, without the need for explicit, vocal negotiations. Its objectives were: (i) avoiding rewriting ad-hoc algorithms for common cases such as two or more NPCs having to cross a small door, going thru a narrow bridge, queuing at a counter, and others were humans commonly coordinate silently; (ii) introducing cultural and contextual sensitivity; (iii) supporting the agent paradigm by adopting a non-centralized approach which, in the long term, will allow to integrate also human players. The result of this work is the INCA framework [10] [11], which exploits DICE's scheduler and introspection facilities and PRESTO's qualities. DICE's introspection has enabled reasoning on the actions being performed by the NPC, while PRESTO qualities have simplified the process of intention recognition. Coordination information is modelled at meta-level, i.e. independently of the actual actions they express which meta-action is currently performed (one of approaching, waiting, engaging), on each resource on which agents are coordinating; the ground actions may be as diverse as, for instance, confusedly crowding rather than orderly queuing in front of a gate when waiting, opening a door rather than pushing a button on a food distributor when engaging, and so on. The meta-level model reduces the coordination problem to determining which is the next NPC that can perform the engaging meta-action; this is decided autonomously by each agent by applying a non domain-specific policy. An agent uses its NPC's perceptions to know with whom it has to coordinate. If all NPCs involved in a coordination adopt the same policy and have the same information, coordinated behavior will result. Importantly, partial information (e.g., not seeing another NPC) and different policies (e.g. strictly FIFO vs "FIFO but with precedence to women and the elderly" vs "ignore everybody else") may cause inconsistencies among agents, as in real life; these are typically solved by handling lower-level issues such as hitting obstacles. The policy for a coordination problem is chosen dynamically; a default one is specified by the modeler to represent the culture of a character but considerations about the internal state of the agent, e.g. the level of the "fear" moderator mentioned above, or the current context, e.g. being on the street rather than within a hospital, may lead to the adoption of a different one.

Of course, avatars controlled by human players by means of traditional HCI devices do not publish the INCA-specific meta-information, which is perhaps easily deducible by humans but nearly impossible algorithmically when relying only on visual perceptions of 3D models. The approach currently being investigated tackles a much larger problem: non-gamer-friendly HCIs for 3D virtual reality, an important issue in the EMT courses held by Delta where trainees are often neither tech-savvy nor young. Rather than operating (potentially many and distracting) input devices to eventually simulate what a puppeteer does with a puppet, the player should "tell" his avatar what to do via simpler, adaptive and context-sensitive interfaces, possibly at a very high cognitive level. In the current experiments [6] [7], a DICE agent completely controls the player's avatar but shows to the player a set of buttons corresponding each to a goal of the agent's prevailing role; mouse-clicking to select objects or locations in the VR allows to filter goals and fill up their parameters. This UI approach essentially delegates most lower-level activities to the agent and leaves decision-making to the player, pretty much at the same abstraction level at which DICE Parts should be (ideally) developed. One of its side effects is the use of INCA by the avatar-controlling DICE agent, thus automatic coordination with NPCs ensues. Research is needed to understand if this approach to UI improves immersivity and thus training effectiveness (since the player needs not to focus on low-level controls) or the opposite (since the player's own identification with her avatar may be partially lost).

## 8 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. Its main design goal was to enable a specialist (e.g. the training strategist of Figure 1) to create or customize sessions for specific training scenarios to be executed possibly with an instructor's supervision. These scripted training sessions 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. 3) 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 the 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.

## 8.1 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 (e.g. the instructor of Figure 1). 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 templates (Sec. 4.1), 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 (a subset of what discussed in Sec. 4.2). 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 invoker. 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 invoker 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.

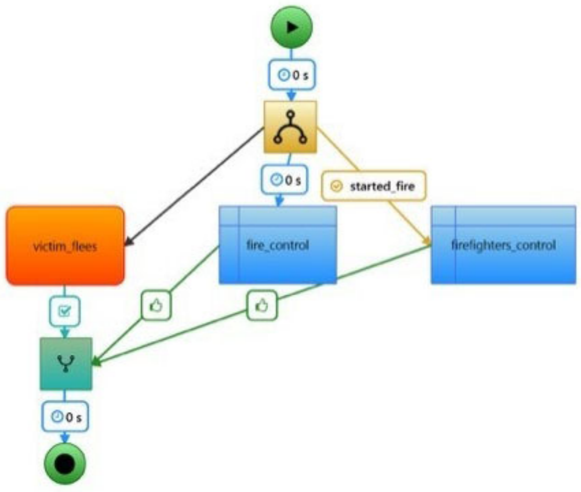
Parallel walks are supported by "fork" scenes. Forks do not submit commands; simply, they visually represent a point from where concurrent paths start. At run-time, all edges from a fork will be followed in the order their respective events occur; this means that all their destination nodes will eventually 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 synchronizing on their conclusion.

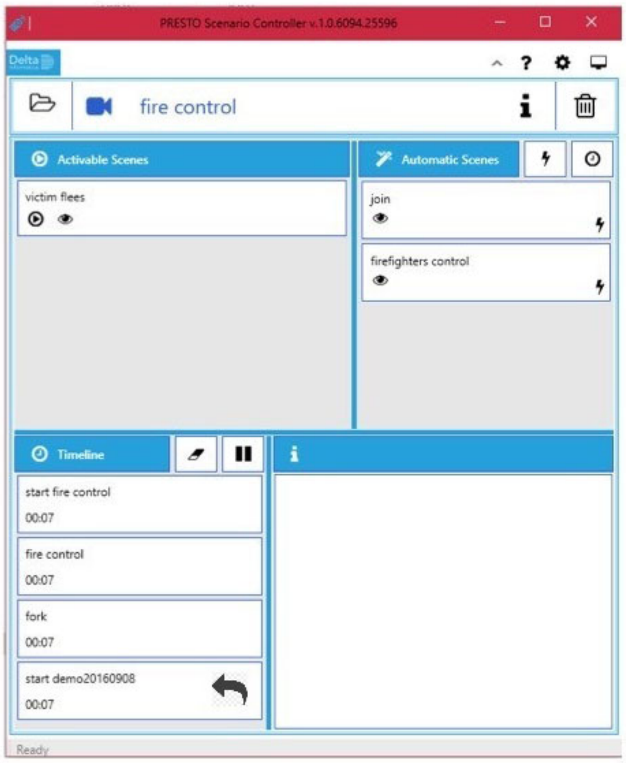
As an example, the left side of Figure 6 shows a snapshot of the graph view of an example of PRESTO Script as shown by the script editor. A fork node (yellow square) is the starting point of three parallel paths: the rightmost one starts a subscript "firefighter_control" when the situation "started_fire" 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) 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 subscripting 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 specifities 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.

Figure 6. Example of script for EMT training (left) and snapshot of the controller (right)





victim flees

join

firefighters control

start fire control
00:07

fire control
00:07

fork
00:07

start demo20160908
00:07

## 8.2 Semantics

Formally, the semantics of scripting can be described as transformations on three sets maintained by the engine: active scripts, edges waiting to be crossed and active situations (i.e. whose truth value is true). These sets are global because they concern the entire game in progress; in other words, PRESTO Scripts should be analyzed as sets of cooperating procedures rather than in insulation.

$AS = \{a|a$ is a currently active script $\}$

$WE = \{w_{AS}|w_{AS}$ is an instance of an edge from $AS\}$

$Sits = \{s|s$ is a currently true situation $\}$

Transformations are the application of a function $F_s$ at the occurrence of an event $E$ that generates a new set of edges waiting to be crossed, of situations and of active scripts according to the rules described in the previous section:

$F_s(E, WE, Sits, AS) \rightarrow \langle WE, Sits, AS\rangle$

Indeed, at the occurrence of an event, one or more edges may be removed, one or more scenes may be processed and this, in turn, may cause other edges to be added as well as situations to be declared true or false. Scenes may cause scripts to start or terminate. Remember that delegating goals to NPCs and changing properties in the virtual environment are (conceptually) instantaneous and have no direct effect on the state of the scripts.

Events are inputs to scripting and are generated in bursts from various sources, thus transformations are applied by taking events one at the time in their order of generation. Focusing on situation changes (which may have been caused by scripting itself as well as being effects of a game's progress), a function $F_e$ computes a set of events by comparing a set $Sits_0$ valid at time $t_0$ and $Sits_1$ at time $t_1$, each event reporting a situation for which the truth value is changed:

$F_e(Sits_0, Sits_1) \rightarrow \{E_1, ..., E_n\}$

Operationally, the algorithm at the core of the script engine maintains a queue of incoming events and the list of instances of edges from the currently active scripts waiting for events to occur and manipulates the tuple space of situations (maintained by the situation engine described in Sec. 4). 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.

A simplified outline of the algorithm is in Figure 7. 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.

## 8.3 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,

```
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
```

Figure 7. Core scripting algorithm

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 facilities that shows how scripts invoke each other, which situations they share, and other details useful for understanding dependencies.

The PRESTO Script Engine, implementing the logic described above, 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 controller programs, which can run on different devices from where the virtual environment is executing. Both controllers allow their users 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 to restore it to a past checkpoint scene. The right side of Figure 6 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 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.

Future work on the implementation will include the integration of the Script editor with the DICE Part editor and other UIs required by PRESTO to support a full-round development environment. Future research topics include improving unsupervised script execution especially for training, e.g. by using player performance parameters to drive scripts. Also, PRESTO Script is being considered for domains other than serious games, in particular decision support and automation of procedures as a complement to ECA (Event Condition Action) rules. For instance, PRESTO Script may be appropriate for reactions to alarms when the results of explorative actions or further events may determine how to proceed. As an example, a "too many connections" alarm by a load-balanced cloud service may immediately start an additional instance on a new server; however, if the alarm doesn't terminate after a while, it may be that a DDoS attack is in progress and further actions or human supervision are required. This try-something-and-see-what-happens logic could easily be programmed in PRESTO Script.

## 9 Pilots

Delta Informatica, in strict collaboration with the APSS (Azienda Provinciale Servizi Sanitari, i.e. public health services department) of the Province of Trento, Italy, has used XVR with PRESTO to create a fire management course for hospital staff. The course is delivered in a highly interactive format by an instructor that shows the XVR screen via a projector, walks and operates in the VR while trainees need to take note of what they see and interactively answer to questions or suggest actions to the trainer while the situation unfolds. Various editions of this course (which takes some 4 hours between theory and practice in VR) have been run in two different hospitals, with some 400 trainees to date; in the after-course questionnaires, trainees systematically reported a high level of satisfaction and higher engagement with respect to traditional courseware. Since trainees do not have to operate on XVR by themselves, HCI difficulties that non-videogame-savvies may experience, especially given the short duration of the course, are avoided. PRESTO has been used mainly to automate manipulations of the VR, otherwise too complex to perform via the native XVR mechanisms, and to introduce relative simple but plausible behaviours of passers-by and teammates in place of the simple animations on fixed paths supported by XVR. Overall, a dozen NPCs and a handful of scripts (handling physical models such as smoke diffusion in addition to progressing a training session) are in use. Following customer contacts and emerging requirements, other courses for emergency services are being prepared, some as first-person, unsupervised serious games. One of the objectives is to address the training requirements of commanders supervising an intervention, which in turn requires "smart" NPCs to play the role of team members and other actors in complex scenarios.

PUG was used as practical of the Agent Oriented Software Engineering courses held at the Computer Science department of the University of Trento in 2015 and 2016. In 2015, some 20 students had to model the autonomous behaviours of different NPCs. The objective was to implement the coordinated procedures adopted by three types of professionals in a health facility (receptionists, nurses, doctors) to handle a flow of patients. The goal of the simulation was to maximize income as computed by the game logic according to the number of professionals involved and patients successfully handled. Behaviours implementing those procedures were written as DICE Parts, exploiting the base behavioral models provided with PUG. In 2016, another 20 students had to write PRESTO Scripts to handle a few types of accident situations (fire breaking in different places) within the setting of 2015 but only with pre-built non-reactive NPC behaviours. Purpose of the simulation-exercise was to prevent the death of the NPCs (automatically determined by the game logic according to smoke intake) by making the professionals act coordinately and in strict accordance with (very generic) safety procedures specified as requirement of the exercise. In both years, after a few introductory lessons to JACK, serious games and PRESTO, the lab consisted of 4 1,5-hour sessions on the editors and PUG; the final assignment was given at the end of the course and the students had two weeks to deliver their projects. The work of each student was marked as part of the final examination and evaluated both according to the results of the simulations and to the readability of the developed parts or scripts. Distribution of marks (including a few non-passes) was according to expectations for this type of course; top achievers demonstrated good creativity in their solutions. On a course satisfaction questionnaire, all participant reported the experience as positive. From the perspective of PRESTO, these labs have been excellent tests of the entire environment, highlighting its potentiality and limitations; worth noting, from a performance perspective, that the best simulations had up to a few tens of NPCs running on average student notebooks. On the minus side, the participants, being computer scientist master students, were too familiar with computing concepts to consider the results of the usability evaluations, especially on the part and script editors, valuable indications for their use as end-user development tools.

## 10 Methodological considerations: modeling sessions, behaviors, emotions

As described above, PRESTO covers many aspects of game and simulation development, including some that are quite novel to gaming if not to the entire software industry: semantic-based, goal-driven, reusable behavioral modelling; cognitive / emotional state evolution; training session / game design around the concept of "situation" and with the ability of a director to intervene on the fly.

A top-down analysis and design methodology for EMT scenarios in an open-world game (such as XVR, where there is no logic controlling progress) has been defined according to a classic user-center approach. By means of interviews to domain experts and trainers, training objectives are spelled out (both positive, e.g. the procedures a trainee should memorize, and negative, e.g. experiencing consequences of errors), scenarios identified (including virtual environments and objects within them), and the unfolding of training sessions written in the form of stories, specifying what each actor (trainee and NPC) is supposed to do. From the stories, specifications of high-level behaviours of NPCs are derived, in forms of team procedures and individual goals, significant events and main procedures to be applied. A number of templates and guidelines have been produced to guide the analysts through this process.

Specifications are eventually given to the development team to produce VR environments, agents' roles and behavioral models, situations and scenario scripts. No methodology has been defined yet for this activity. At the moment, work is driven by experience as well as by the available library of reusable assets that may substantially influence implementation choices. The major issue to be tackled is the significant gap between high-level decision-making as captured with the process described above and the practicalities of development on top of existing engines, requiring even the most common human action to be designed starting from animations and other graphic effects. The machinery offered by PRESTO in terms of semantics and reusable behaviours should help closing this gap over time while models are incrementally developed. However, this work implies a significant effort in defining abstractions applicable over multiple scenarios and types of NPCs. For instance, the pilots presented above, even if used in different game platforms, share the same basic roles concerning navigation (including relatively complex goals such "flee to a certain type of location", "follow that character") and certain common actions (taking, pushing and pulling certain types of objects, pushing buttons, emergency actions such as operating fire extinguishers, and so on).

As mentioned in Sec. 5, research has been spent to take into consideration the emotional aspects of a story. In short, this consists in annotating each situation of a story within a matrix where, for each major event of a story captured as above, PAD (Pleasure / Arousal / Dominance) values are given for each involved character, according to its professional role and capability. Additionally, alternative reactions should be specified according to the current emotional state of the character. A computational model has been defined so that, when an event occurs (triggered by a situation being asserted), it takes in account moderators (fear and fatigue in the specific examples) and situation-specific PAD values to calculate the updated emotions and moderators values; this determines the behavioral profile to apply, which in turn contains the appropriate reaction to the event. More research is needed to move from current proof-of-concepts to reusable models of emotion and a methodology for domain experts rather than psychologists, for instance using fuzzy categories for PAD values (currently in the range [-1,1]) or classifying events appropriately e.g. as "scary" or "comforting".

## 11 Conclusion

PRESTO covered a lot of ground in the area between training conception and serious game development with current virtual reality and 3D game engines. Its contributions are mainly in the area of semantics for decision-making, NPC behavior modelling technology, scenario scripting. Some of the open research questions are purely technological but most concern modelling methodologies, for instance concerning the definition of reusable roles and the influence of events on emotions. Some preliminary work towards a non-gamer-friendly UI, by which a player controls an

NPC accepting high-level commands, has been done, with the side effect of simplifying NPC / player coordination. PRESTO is not available off-the-shelf but Delta Informatica is willing to discuss research collaborations or commercial exploitations. PRESTO was partially funded by a grant of the Provincia Autonoma di Trento (PAT), Italy.

## References

[1] P. Calanca and P. Busetta, "Cognitive Navigation in PRESTO," in *AI&Games workshop @ AISB 2015*, Canterbury (UK), 2015.

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

[3] F. Ritter, J. Bittner, S. Kase, and R. Evertsz, "CoJACK: A high-level cognitive architecture with demonstrations of moderators, variability, and implications for situation awareness," *Biologically Inspired Cognitive Architectures*, vol. 1, pp. 2–13, 2012.

[4] R. Evertsz, M. Pedrotti, P. Busetta, and H. Acar, "Populating VBS2 with realistic virtual actors," in *Proceedings of the 18th Conference on Behavior Representation in Modeling & Simulation (BRIMS)*, Sundance Resort, Utah, 2009.

[5] R. Evertsz, F. E. Ritter, P. Busetta, M. Pedrotti, and J. 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*, 2008, pp. 80–89.

[6] M. Palma, "Studio e progettazione di un prototipo di Serious Game giocabile in prima persona utilizzando il Framework PRESTO," Tesi di Laurea Triennale in Informatica, Università degli Studi di Napoli Federico II, 2016.

[7] F. Orioli, "Una Interfaccia Punta e Clicca per l'interazione tra Unity3d e PRESTO," Tesi di Laurea Triennale in Informatica, Università degli Studi di Napoli Federico II, 2016.

[8] 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.

[9] 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): Springer LNCS, 2015.

[10] M. Robol, P. Giorgini, and P. Busetta, "Applying social norms to implicit negotiation among Non-Player Characters in serious games," in *Proceedings of WOA 2016*, Catania, 2016.

[11] M. Robol, P. Giorgini, and P. Busetta, "Applying social norms to high-fidelity pedestrian and traffic simulations," in *International Smart Cities Conference (ISC2)*, Trento, 2016.

[12] D. D. Corkill, "Blackboard systems," 1991.

[13] P. Busetta, C. Ghidini, M. Pedrotti, A. 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.

[14] J. Boyd, "The Essence of Winning and Losing," *A Discourse on Winning and Losing*, no. August, 1987.

[15] C. Becker, S. Kopp, and I. Wachsmuth, "Simulating the emotion dynamics of a multimodal conversational agent," *Affective Dialogue Systems*, no. 2, pp. 154—-165, 2004.

[16] Z. Menestrina, A. D. 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.