Service Composition with PDDL Representations and Visualization over Videogame Engines

Giuseppe De Giacomo* Valsamis Ntouskos* Fabio Patrizi† Stavros Vassos* Davide Aversa*

*DIAG - Sapienza University of Rome
Rome, Italy
Email: {degiacomo, vassos, ntouskos, aversa}@dis.uniroma1.it

†KRDB - Free University of Bozen-Bolzano
Bozen-Bolzano, Italy
Email: patrizi@dis.uniroma1.it

***Abstract*—We devise a succinct knowledge representation framework based on a nondeterministic variant of a well-known Artificial Intelligence formalism, called PDDL, for representing dynamic domains in Planning. We represent the environment and the agents' (high-level) behavior as distinct PDDL action domains and exploit service composition techniques, to compose agent behaviors so as to realize a collective behavior of interest to the user. Specifically, we characterize the computational complexity of the problem and give effective algorithms for solving it, taking advantage of the succinct representation in PDDL. We explore the visualization of service composition over this framework based on the videogame metaphor of virtual worlds using a popular engine. The execution then of agents as game characters provides a form of procedural attachment of atomic actions to concrete interactions within a realistic 3D space.**

I. Introduction

The goal of the line of research followed in this paper is to explore videogame technology as a setting for studying the new generation of interactive spaces. Smart spaces, virtual environments, and mixed-reality applications are becoming more ubiquitous and sophisticated. In such applications devices and agents, which can be either real or virtual, behave and interact with users in a complex way. Here, we explore the idea of introducing succinct knowledge representation elements as a high-level layer for enabling advanced forms of service composition for components in these environments.

We consider a standard videogame engine provided with models for the physical world and non-player characters (NPC) acting in it. We see NPCs as agents, operating in a shared virtual environment, which export their behavior as conversational services. To formally model them, we devise a knowledge representation framework, e.g., based on a planning domain formalism [1] and action theories in AI [2]. NPCs are thought of as agents whose behavior is modeled in terms of *high-level* actions that represent "physical" action and interaction points. Note that such high-level actions correspond in general to complex operations. For instance, a vacuum cleaner could be instructed to execute a clean operation, which includes moving around in a room, sensing dust, and cleaning dirty areas.

In other words, we map *atomic high-level actions* of agents into *complex low-level behaviors* that are influenced by the sensors and actuators on board of the agents, as well as by the "physical laws" they must obey in the virtual space of the game engine environment. This can be thought of as a sort of concrete *procedural attachment* [3] to the high-level actions. Our intention then is to study the new generation of interactive smart spaces in virtual environments that simulate the low-level details and also allow for user-friendly human interaction with the virtual world and devices as agents.

In particular, we represent the environment as an action domain, expressed in a nondeterministic variants of the well-known formalism PDDL [4], [5], which describes preconditions and effects of actions in terms of fluents, i.e., predicates whose value changes from state to state. We also use preconditions and effects to compactly describe the high-level behavior of each agent in terms of how its internal state evolves, again using nondeterministic PDDL. Agent actions are coupled with environment actions so that they have an effect on the environment. Using this framework we study how to *compose* agent behaviors so as to realize a desired *collective behavior* of interest to the user, by exploiting recent literature on service composition [6], [7], [8], [9].

Notably, differently from the literature above, here we represent both environment and agent behavior succinctly, taking advantage of PDDL. In spite of this, we show that composition remains EXPTIME-complete even though the action domain and the behavior of agents are compactly expressed in terms of actions and fluents. We implemented the framework in the game engine Unity (unity3d.com), where the execution of agents as NPCs provides a form of procedural attachment of atomic actions in the specification to concrete interactions, within a simulated realistic 3D space, as well as under the effect of humans that may participate as players in the traditional sense or through mixed reality interfaces.

II. Virtual Environments and Game Engines

Perhaps the most successful example of virtual environments are video-games. The video game industry is driving globally an enormous amount of human players of all ages to spend many hours per week in virtual game-worlds interacting with virtual devices and characters through many platforms. One interesting feature of the videogame engines, on which these virtual environments are built, is that they offer a powerful development framework and an execution engine which allows one to easily implement virtual agents and run them in a very realistic environment that accurately simulates real-world lights, movements, and general physics, among other things.

From an abstract point of view, these game engines provide means to describe (*i*) a *game-world* that simulates a "physical"

space, subject to precise physics laws; (*ii*) *game-objects* that lie in the space such as walls, stairs, containers, that interact following the physical laws of the space; and (*iii*) *non-player characters (NPCs)*, essentially autonomous agents that can be programmed to follow a desired behavior.

Non-player characters are special game-objects. On the one hand, they are complex objects with joints and several degrees of freedom, e.g., humanoids or robots or complex mobile machines, subject to physical laws of the space; on the other hand, they are equipped with a set of *physical behavioral capabilities* such as moving, standing, picking up objects, aiming, shooting, each of which is connected to an appropriate animation rendered by the game engine.

NPCs are situated in the game-world in the sense that they can see and act based on their *field of view* that provides a space for interacting with the environment. For example, an NPC may be informed of game-objects entering their field of view which may trigger a reactive response. Essentially, each NPC features a repository of programmed routines that describe *conceptual atomic actions* that can be performed during the execution of the NPC in the game-world. These actions can be used to program complex behaviors that describe the lifetime of NPCs, typically following reactive paradigms such as finite state machines [10] and behavior trees [11].

We may consider the representation as formed by two levels: a *low-level* that handles the game-world, game-objects, and NPCs at the level of the physics, and what we may call a *middle-level* which is a sort of *symbolic account* of the game environment. This essentially includes a network of *points of interest*, the conceptual properties of points of interest and game-objects, and the repositories of conceptual atomic actions of NPCs. This middle-layer representation is in fact *implicit* in the details of the game engine data structures and programs in the implementation of the game setting. What we do here is to systematize this middle-level into a *high-level* based on knowledge representation principles focusing on an action-driven account of NPCs, that can be used for principled deliberation based on automated reasoning (and in particular in this work, based on agent behavior composition).

We consider as objects relevant game-objects, NPCs and points of interest. We map relevant properties over these objects as fluents, i.e., predicates that may change their truth value during execution. We map relevant conceptual atomic actions in the repositories attached to NPCs to actions over these fluents. A natural separation between *common fluents* and *private fluents* for each NPC arises, emphasizing the distinction a *shared environment* and *internal stateful behaviors* for NPCs. Actions are described in terms of preconditions and effects over the fluents of the shared domain as well as preconditions and effects over the internal states of NPCs.

Note that the representation is through the eyes of the designer who has complete information over the state of the shared environment and the internal states of NPCs. It is certainly interesting to attach knowledge directly to NPCs but this is not in the scope of this work. Observe also that this high-level domain is in fact finite, hence the representation can be propositional. This enables us to use the standard knowledge representation formalism of the Planning Domain Definition Language (PDDL) [12] as the basis of our formalization. Finally, observe that we do not use PDDL for planning purposes, but as a concrete formalism for compactly describing action domains and NPCs behavior.

Nonetheless we need to consider two sources of nondeterminism for the high-level representation. First, conceptual actions of NPCs may involve randomized outcomes, for example examining a facility may give different readings. Second, as many of the low-level details are abstracted, even an atomic conceptual action that in the low-level is deterministic, in the high-level may become nondeterministic. For example, moving from one location to another may involve the discharge of battery levels for an NPC robot, which is precisely handled in the low-level representation of the game setting, but may be abstracted to simply three states of "low/medium/full".

III. Knowledge Representation Framework

As high-level action description formalism we use a nondeterministic variant of PDDL, formalized as follows. An *action domain specification* is a tuple $\mathcal{D} = (P, s_0, Act)$, where:

- P is a finite set of propositions;
- $s_0 \subseteq P$ is the domain's initial state;
- Act is a finite set of actions, each specified by an expression of the form $[a : \varphi, \eta]$, where:
  - a is the *action name* (unique in $\mathcal{D}$)
  - φ is the *action precondition*, i.e., a propositional formula over P, and
  - η is the *nondeterministic conditional effect* of a, i.e., an expression of the form $(\texttt{oneof}\ ne_1, \ldots, ne_n)$ with each ne_i a list $(ce_{i1}, \ldots, ce_{in_i})$ of expressions ce_{ij} of the form $(\texttt{when}\ \psi_{ij}, (add_{ij}, del_{ij}))$, where ψ_{ij} is a propositional formula over P and $add_{ij}, del_{ij} \subseteq P$.

Essentially, $\mathcal{D}$ represents an action domain in the usual sense, where actions have nondeterministic conditional effects.

For example, consider a domain with a number of patrolling robots and binary predicates $At(r, loc)$ and $Battery(r, lev)$, representing, respectively, the location and the battery level of each robot. We adopt the typical prefix notation of PDDL and denote these as `(at ?r ?loc)` and `(battery ?r ?lev)`, where `?r`, `?loc` and `?lev` are variables, denoted in PDDL with a preceding question mark. Assuming a finite number of robots, e.g., `r1`, `r2`, locations, e.g., `loc1`, `loc2`, `loc3`, `loc4`, and battery levels, e.g., `high`, `med`, `low`, and no other predicates, a corresponding set of propositions P for this domain can be defined as the set of all combinations of ground instances for predicates `(at ?r ?loc)` and `(at ?r ?loc)` for appropriate objects, including for instance `(at r1 loc1)`, `(at r1 loc2)`, etc, and `(battery r1 high)`, `(battery r1 low)`, etc.

A state is a set of propositions that describe the state of affairs in the domain. For instance, the initial state s_0 for this simple domain could be {`(at r1 loc1)`, `(at r2 loc1)`, `(battery r1 full)`, `(battery r2 full)`}, representing that both robots are fully charged and located at `loc1` . Actions operate on a state by looking into the propositions the state consists of and manipulating them using "templates" based on variables: the action precondition specifies whether the action is executable in a state, and effects of the action specify how the state is transformed after the execution of the action. For

example, consider an action $goto(r, oldloc, newloc)$, denoted in PDDL as `(goto ?r ?oldloc ?newloc)`, which is intended to represent that robot `?r` moves from location `?oldloc` to location `?newloc`. The following is a PDDL specification of the intended effect according to the previous definition:

```
[(goto ?r ?oldloc ?newloc) :
     (not (battery ?r low)) ,
     (when true (and (at ?r ?newloc)
                     (not (at ?r ?oldloc))))]
```

In this action there is the precondition that the robot should not have low energy and two (deterministic) effects, namely that the proposition stating the old location is removed from the state description and the new location is added; the negative effect `(not (at ?r ?oldloc))` is denoted with logical negation and is considered to be in the "delete list" del of the action, while the positive effect `(at ?r ?newloc)` is considered to be in the "add list" add of the action.[1]

The domains we consider allow also for nondeterministc effects of actions, therefore have a particular structure in order to specify each of the potential effects be means of specifying a set of potential effect specifications. These are denoted by using the `oneof` notation for a collection of conditional effect description like the one we specified in the `goto` action, e.g., a repair action that potentially drops the battery level of a robot when performed. We will see such an example later.

Formally, a state s is a subset of P associated with the propositional assignment v_s s.t. $v_s(p) = \top$ iff $p \in P$. A transition $s \xrightarrow{a} s'$ for action $[a : \varphi, \eta]$ in state s exists iff: a is executable in s according to its precondition, i.e., $v_s \models \varphi$; and there exists some $ne_i \in \eta$ s.t. $s' = (s \setminus del_i) \cup add_i$, for $del_i = \bigcup_{\{ce_{ij} \in e_i | v_s \models \psi_{ij}\}} del_{ij}$ and $add_i = \bigcup_{\{ce_{ij} \in e_i | v_s \models \psi_{ij}\}} add_{ij}$. Intuitively, when action a is executed, one of its nondeterministic effects ne_i is nondeterministically chosen, then all of its conditional effects ce_{ij} whose condition ψ_{ij} is satisfied in the current state are selected, and, for all of these, the propositions in del_{ij} are removed from the current state, and those in add_{ij} are added. Deletions are performed first, at once.

We appeal to this compact way in order to model the high-level shared environment and agent behaviors as action domains. We assume a system $\mathcal{S}$ consisting of a domain $\mathcal{E} = (P_\mathcal{E}, s_{\mathcal{E}0}, Act_\mathcal{E})$, representing the shared environment, and n distinct domains, representing the behavior of the agents acting in $\mathcal{E}$, $\mathcal{B}_i = (P_i \cup P_\mathcal{E}, s_{i0}, Act_i)$, s.t. $P_i \cap P_\mathcal{E} = \emptyset$ and $names(Act_i) \subseteq names(Act_\mathcal{E})$, with $names(Act)$ denoting the set of action names occurring in a set of actions Act. Notice that agents have a set P_i of local propositions which accounts for their internal states.

The executability and the effects of actions of an agent in an environment $\mathcal{E}$ depend on and affect both the state of the agent and that of the environment. For instance, in a blocks-world scenario, an agent that has the arm ready to pick up a block (precondition on local state), can do so only if the block is free (precondition on environment state). Similarly, executing an action yields effects on both the agent and the environment, according to the respective specifications. To capture this, we define the *execution of a behavior $\mathcal{B}_i$ on the environment $\mathcal{E}$* as the transition system $D_i^\mathcal{E} = (\Sigma_i^\mathcal{E}, \sigma_0^\mathcal{E}, \rightarrow_i^\mathcal{E})$, where:

- $\Sigma_i^\mathcal{E} \subseteq 2^{P_\mathcal{E}} \times 2^{P_i}$ is the set of states of $D_i^\mathcal{E}$;
- $\sigma_0^\mathcal{E} = (s_{\mathcal{E}0}, s_{i0})$ is the initial state of $D_i^\mathcal{E}$;
- $\rightarrow_i^\mathcal{E} \subseteq \Sigma_i^\mathcal{E} \times names(Act_\mathcal{E}) \times \{1, \ldots, n\} \times \Sigma_i^\mathcal{E}$, is the transition relation of $D_i^\mathcal{E}$, s.t. $\sigma \xrightarrow{a,i} \sigma'$ iff:
  - $\sigma = (s_\mathcal{E}, s_i)$ and $\sigma' = (s'_\mathcal{E}, s'_i,)$;
  - there exists an action $[a : \varphi_\mathcal{E}, \eta_\mathcal{E}] \in Act_\mathcal{E}$ and an action $[a : \varphi_i, \eta_i] \in Act_i$ s.t. $s_\mathcal{E} \xrightarrow{a} s'_\mathcal{E}$ and $s_i \xrightarrow{a} s'_i$.

Notice that every agent action is paired, through the name, with (exactly) one in the environment. Such pairing accounts for the causal relationships between the environment and an agent, when the latter executes an action. Observe, however, that the effects of actions on the environment, do not depend on the specific agent that executed the action: as far as the shared environment is concerned, *some* agent will execute a given action. We stress that agents are the only entities able to execute actions, and thus trigger changes in the environment.

IV. Agent Behavior Composition

Next we turn to *agent behavior composition* [6], [7], [9]. We define a target behavior $\mathcal{T}$ over an environment $\mathcal{E}$ as $\mathcal{T} = (P_\mathcal{T} \cup O_\mathcal{E}, s_{\mathcal{T}0}, Act_\mathcal{T})$. The target behavior represents the desired behavior that a user would like to interact with (over $\mathcal{E}$), which in general is not available in $\mathcal{S}$. Notice that we allow for nondeterminism in the target behavior assuming that the client executing the target can actually choose not only the action to execute next but also the next state in the target behavior, like in [8]. In this way we can model decisions that the client can make because of the result of actions. Notice that all results in [7] continue to hold in this extension.

The goal of behavior composition is to *build an executable controller* that coordinates the available behaviors in an appropriate way, so as to *simulate* the execution of the target behavior over $\mathcal{E}$. [7] formalizes this intuition and shows a solution based on the notion of *simulation* [13]. This technique relies on a fixpoint computation that returns a finite-state automaton, called the *controller generator* (CG), which is able to generate at runtime any controller that can make $\mathcal{S}$ evolve so as to simulate the execution of $\mathcal{T}$ over $\mathcal{E}$.

Specifically, given the system $\mathcal{S}$ and a target behavior $\mathcal{T}$ over $\mathcal{E}$, let $D_\mathcal{T}^\mathcal{E}$ and $D_i^\mathcal{E}$ be the executions of $\mathcal{T}$ and of $\mathcal{B}_i$ over $\mathcal{E}$, respectively. Moreover, let $s = (s_\mathcal{E}, s_1, \ldots, s_n) \in 2^{P_\mathcal{E}} \times 2^{P_1} \times \cdots \times 2^{P_n}$ be a state of the whole system $\mathcal{S}$, and $t = (s_\mathcal{E}, s_\mathcal{T})$ a state of $D_\mathcal{T}^\mathcal{E}$ (notice that $s_\mathcal{E}$ matches in s and t). Then, s is said to *ND-simulate* t, written $s \preceq_{ND} t$, iff for every transition $t \xrightarrow{a} t'$ in $D_\mathcal{T}^\mathcal{E}$, with $t' = (s'_\mathcal{E}, s'_\mathcal{T})$, there exists a behavior $\mathcal{B}_i$ such that:

- there exists a transition $(s_\mathcal{E}, s_i) \xrightarrow{a} (s'_\mathcal{E}, s'_i)$ in $D_i^\mathcal{E}$;
- for every such transition, it is the case that $(s'_\mathcal{E}, s_1, \ldots, s'_i, \ldots, s_n) \preceq_{ND} (s'_\mathcal{E}, s'_\mathcal{T})$.

Finally, $\mathcal{S}$ is said to *ND-simulate* $\mathcal{T}$ (on $\mathcal{E}$), written $\mathcal{S} \preceq_{ND} \mathcal{T}$ iff it is the case that $(s_{\mathcal{E}0}, s_{10}, \ldots, s_{n0}) \preceq_{ND} (s_{\mathcal{E}0}, s_{\mathcal{T}0})$.

Intuitively, starting with system $\mathcal{S}$ and target behavior $\mathcal{T}$ in their respective initial states (observe that this fixes also the

[1] As the specification of effects requires that they are formed as conditional effects in the general case, here we also followed the `(when condition effects)` pattern to denote the effects of this action using `true` for the condition and the logical `and` connective for the effects.

state of the environment), for any transition with a given action that $\mathcal{T}$ can execute, based on its current state and that of the environment, there exists some behavior $\mathcal{B}_i$ that can execute the same action, and that, no matter how $\mathcal{E}$ and $\mathcal{B}_i$ happen to evolve, this property propagates to the successor states of the target behavior and the system.

Next, we provide some insights on the execution of the CG. Essentially, the CG is a finite-state transducer interposed between the client and the behaviors, whose task is to receive target transitions requests from the client and appropriately delegate the execution of the action therein to agents. This can be recorded as a look-up table to be used at runtime to execute the CG. Assume that, in the current state s of $\mathcal{S}$, the client requests an transition $t \xrightarrow{a} t'$ executable in the current state t of $\mathcal{T}$. Then, the CG returns one among the indices in the table associated with s, t, t' and a, i.e., the indices of the behaviors that can execute a while guaranteeing the possibility of serving all future action requests compliant with $\mathcal{T}$. Let $\mathcal{B}_i$ be the selected behavior. After executing a, $\mathcal{B}_i$ and $\mathcal{E}$ (nondeterministically) move to a possible successor state. The CG then progresses the state of $\mathcal{T}$ to t'. When a new target transition is requested, the CG observes the current state of the system $\mathcal{S}$, selects a new behavior for executing the action (which is guaranteed to exist given the previous choice), the system progresses again, the CG progresses the target behavior, and a new iteration can take place.

Notice that [7] uses an explicit representation for the involved behavior and the environment, while here we adopt a compact representation formalism for a slightly generalized problem. It is interesting then to understand the complexity of computing the ND-simulation and the CG in our case.

Theorem 1: Given a system $\mathcal{S}$, consisting of an environment $\mathcal{E}$ and a set of available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$, and a target behavior $\mathcal{T}$, all compactly represented as action domains, the largest ND-simulation relation between $\mathcal{S}$ and $\mathcal{T}$ can be computed in time polynomial in $2^{|P_\mathcal{E}|+|P_1|+\cdots+|P_n|+|P_\mathcal{T}|}$.

In other words we are exponential in the size of the representation. Notice this is the same complexity as e.g., model checking in terms of compact representation, the so call "program-complexity", see e.g., [14]. Interestingly the original problem in [7] which use explicit representation of states (exponentially larger than a compact representation) is also exponential, though only in the number of available behaviors. Notice that in our compact formalism, the size of the state space of each structure is (at most) exponential in the number of propositions. Thus, with a compact representation formalism, one might expect a further exponential blowup in the complexity. However, this is not the case, as the above theorem shows. Finally, notice that the composition problem is EXPTIME-complete already for much simpler cases [15].

V. The Power Plant Surveillance Scenario

We consider a case of a hypothetical power plant and a surveillance scenario with n multi-purpose robots that are able to navigate to a number of designated areas, and perform security-related operations. In this scenario, the power plant is the virtual environment possibly simulating a real setting. The atomic conceptual actions of the NPC robots are (*i*) modeled mathematically as available behaviors in the behavior composition framework that we described in the previous section, and (*ii*) implemented as NPCs in the 3D game world provided by a game engine. As we will be relying on the Unity game engine for our implemented system, for illustration purposes we adapted the 3D game-world of the action mini-game "Angry Bots" (unity3d.com/gallery/demos/live-demos#angrybots) to act as the power plant simulated virtual environment.

The modeling is done from the point of view of a human security manager who receives information from the NPC robots and decides on next operations to be carried out based on a predefined security protocol. There are four points of interest as far as the NPC robots and the target protocol are concerned, namely rooms A, B, C and a charging station CS. The NPC robots are able to move to these areas and perform a measurement of the radiation levels. Under certain circumstances the manager may instruct an adjusting operation in the equipment of the area.

Each of these robots is self-powered using batteries. Battery levels are follow a discrete scale and are represented by the objects `high`, `medium`, `low`. Adjusting the equipment deterministically reduces the battery level of the NPC from `high` to `medium` and non-deterministically from `medium` to `low`. The inspect and move actions on the other hand require little power and for simplification purposes are not considered in the modeling. Recharging is possible by going to the charging station. Note that the characteristics of this scenario are modeled at a different level of detail: (*i*) the high-level as a shared environment and agent behaviors; and (*ii*) the middle-level and low-level as appropriate an implementation in the Unity game engine.

The shared environment has three fluents: `(at ?r ?a)` capturing the location of robots, `(power ?n)` capturing the overall power level of the system in terms of discrete objects for $i \in \{1, \ldots, 2n\}$, and `(alert ?a)` capturing whether area `?a` has high radiation levels. Also objects $\texttt{npc}_i$ for $i \in \{1, \ldots, n\}$ are included in the domain. The actions of the shared environment are the following: `(goto ?a)`, capturing that some robot is instructed to move to area `?a`, `(inspect ?a)` capturing that some robot inspects the radiation levels of area `?a`, `(adjust ?a)` capturing that some robot adjusts the equipment in area `?a`, and `(recharge)` capturing that one robot with low battery is instructed to get to the charge station and recharge. Each `adjust` action deterministically drops the overall power by one, contextually dropping the battery level of the robot performing the action, and representing a pessimistic estimation of the overall power status of the whole system, while every `recharge` action increases it by one (appropriate fluents are used to formalize the ordering of the battery and power levels but are omitted here for brevity). Moreover, each NPC has one internal fluent `(battery ?l)` that holds the internal level in terms of objects `high`, `medium`, `low`.

Consider a target behavior that involves two internal states represented by the fluent `red`. In normal mode the user can request a `(goto ?a)` followed by an `(inspect ?a)` action for some area `?a`, or `recharge`. In the red mode he can request a `(goto ?a)` followed by an `(adjust ?a)` action for some area `?a`, or `recharge`. One changes from normal mode to red mode through the non-deterministic effects of a `inspect` action which includes a decision by the user to move red mode.

In red mode one moves deterministically back to normal mode once all necessary adjusting has been performed.

The PDDL specifications of the main actions are provided in Table I. Observe that the NPCs and the target behavior share the same actions with the environment, however the preconditions and the effects vary to capture the corresponding behavior. Moreover, the fluents of the environment are accessible from all the other behaviors, while the fluents of all the other behaviors are accessible only to them. Also, note that the red mode raised by the target does not depend on the current result of the inspection, but requires that at least one area was in alert mode before executing the action.

It can be shown that the CG can be computed for realizing the target using the available NPCs in the shared environment. For illustration purposes, Table II shows a small part of the CG for the case of two NPCs. The PDDL files and output can be found at jaco.dis.uniroma1.it/#example4.

VI. The System

Our system is based on the game engine Unity (http://unity3d.com), an integrated game developing environment that includes a high-quality physics and rendering engine. For implementing NPCs we chose to realize a basic action-driven architecture. For this architecture we relied on the basic low-level features of Unity related to the "physical" properties of game-objects, i.e., meshes, skeletons, joints, colliders, light shaders, etc, but also introduced a simple sense-think-act architecture in order to model the NPC behavior in terms of named, actions, conditions, and a precise representation for the internal state of the NPC. The implemented architecture consists of four basic components as follows.

- The *perception* component is responsible for identifying objects in the field of view of the NPC, including conditions or events that occur, which can be useful for the deliberation component. In the typical case it is attached to a mesh object surrounding the NPC, e.g., a sight cone positioned on the head of a robotic device.
- The *deliberation* component is responsible for deciding the immediate action that should be performed by the NPC by taking into account the input from the perception component as well as the internals of this component. This component can be used to abstract the behavior that the NPC should follow, which could be for instance expressed in terms of reactive or proactive behavior or in our case a combination of these along with a set of commands that are given as instructions to the NPC.
- The *control* component is going over a loop that passes information between the perception and deliberation components, and handling the execution of actions as they are decided. In particular, the controller is agnostic of the way that perception, deliberation and action is implemented, but is responsible for coordinating the information between the components while handling exceptions, monitoring conditions and actions, and allocating resources to the deliberation component accordingly.
- The *action* component is responsible for realizing the actions that are decided by the deliberation component in the game-world and provide information about the state of action execution, e.g., success or failure.

Fig. 1. A patrolling robot in the virtual environment.

Essentially the perception and action components provide a bridge between the low-level aspect of NPCs and the middle-level of symbolic properties and conceptual atomic actions that were discussed in the beginning of the paper. For the scope of our agent composition framework one can think of the deliberation component as adding a high-level knowledge representation and reasoning layer by means of the computed CG: the deliberation component simply acquires orders corresponding to the user choices through a dashboard that will be described next, following them in a reactive fashion. Note though that each conceptual atomic action may be a complex procedure that handles low-level details or even includes a behavior strategy expressed as a nested deliberation instance.

As far as the power plant scenario is concerned, each of the patrolling robots is implemented as a group of game-objects that model the moving parts of the device also with corresponding meshes and textures. For simplicity a model of a small robot that is available in the Angry Bots project was used. As far as the internal state of the patrolling NPC is concerned, each NPC has a battery level that ranges from 0 to 100. Each time a low-level action is executed, the battery level goes down by a fixed value that depends on the action following the details of the scenario. A snapshot of one of the NPC robots can be seen in Figure 1; visualized on top of the robot one can see both its low-level and its abstracted high-level internal state.

In our system the target behavior is represented as a *stateful dashboard* such at every point in time the available actions are presented as selectable options and the feedback from the system is shown on the screen. This target is actually realized through the controller generator (CG). In particular, when the next action is selected, the system uses the CG as a look-up table that, given the current state of affairs and the chosen action, specifies which NPCs may be selected to realize the chosen action, and randomly picks one of them to execute it (without showing the internal CG options to the user). The NPC is instructed to execute the chosen action, and after action execution is finished, the dashboardd is notified of the resulting state for the NPC and the shared environment and new choices are presented to the user.

A so-called "composer" module is in charge of computing the CG. Actually this component does not need to be embedded to the Unity-based virtual environment, but can be deployed to the cloud and used through a web-service

TABLE I. PDDL SPECIFICATION OF THE ACTIONS FOR EACH BEHAVIOR

	Environment	**NPC**	**Target**
goto ?a			
preconditions:	()	()	()
effects:	()	(forall (?b) (when (at npc ?b) (and (at npc ?a) (not (at npc ?b)))))	()
inspect ?a			
preconditions:	(exists (?r) (at ?r ?a))	(at npc ?a)	(not (is-red))
effects:	(oneof (alert ?a) ())	()	(oneof (when (exists (?b) (alert ?b)) (is-red)) ())
adjust ?a			
preconditions:	(and (alert ?a)(exists (?r) (at ?r ?a)))	(and (at npc ?a) (not (battery low)))	(and (power ?c) (gt ?c N) (is-red))
effects:	(not (alert ?a))	(oneof (and (when (battery high) (and (battery med) (not (battery high)))) (when (battery med) (and (battery low) (not (battery med))))) (when (battery high) (and (battery med) (not (battery high)))))	(and (when (forall (?b) (or (not (alert ?b)) (= ?a ?b))) (not (is-red))) (forall (?c ?d) (when (and (power ?c) (succ ?d ?c)) (and (power ?d) (not (power ?c))))))
recharge			
preconditions:	()	()	(not (power 2N))
effects:	()	(forall (?b) (when (at npc ?b) (and (at npc CS) (battery high) (not (at npc ?b)) (not (battery med)) (not (battery low)))))	(forall (?c ?d) (when (and (power ?c) (succ ?c ?d)) (and (power ?d) (not (power ?c)))))

TABLE II. PART OF THE COMPUTED CONTROLLER GENATOR FOR A SCENARIO WITH TWO NPCS.

Target	Environment	NPC1 (r1)	NPC2 (r2)	Action	Allocation
()	(at r1 a), (at r2 c), (power 2)	(battery med)	(battery high)	(inspect c)	r2
(is-red)	(at r1 c), (at r2 a), (alert c), (power 4)	(battery high)	(battery high)	(adjust c)	r1
(is-red)	(at r1 a), (at r2 c), (alert b), (alert c), (power 2)	(battery med)	(battery med)	(goto b)	r1

interface. This is exactly what happens in our system where the composer is available as a cloud web-service that, given the PDDL specification for a set of available behaviors, a shared environment, and a target behavior, it computes and returns the CG (when one exists). The composer is based on a Java-based synthesis general-purpose engine called JTLV (http://jtlv.ysaar.net), that essentially performs fixpoint computations over *game structures* [16] in a (space-)efficient way, by exploiting symbolic techniques.

VII. RELATED WORK

Regarding simulating smart spaces, there is ongoing work related to virtual environments such as UbiREAL (ubireal.org) [17] which is a framework for building virtual smart spaces, and DIVAS (mavs.utdallas.edu/projects/divas) [18], a plugin-based multi-agent 3D simulation framework for large scale events. Compared to the direction of our work, one observation is that these platforms come from a research community in which the focus is mostly on different aspects such as the generation and analysis of synthetic low-level sensor data. Instead, in this work we aim for composing behaviors of NPCs wrt a desired target process.

As far as video games are concerned, there is a lot of related work for coordinating non-player characters in the context of interactive storytelling [19]. In terms of character autonomy [20] and the strong story / strong autonomy spectrum our approach lies closer to the former end, as NPCs are allowed to act as autonomous entities but only as long as they do not change their internal state captured in the corresponding behavior. Perhaps closest to our AI technique are those approaches that employ STRIPS and HTN planning, such the work of Porteus *et al.* [21], which makes use of PDDL with constraints to characterize the sequencing of events in the intended storyline, as well as systems Mimesis [22], GADIN [23], MIST [24], Zócalo [25], and FAtiMA [26] that employ some form of planning for coordinating NPCs. Nonetheless, there are differences as we explain next.

First note that we use PDDL as a concrete syntax for action theories, e.g., ala [2], exploiting its capability to formalize preconditions and effects of actions. In particular, we use PDDL to model how the environment and the internal states of characters change – not to describe a planning problem. Note also that we extend PDDL to handle nondeterminism in order to model the uncertainty generated by the low-level interactions in the virtual environment.

Second, although we use PDDL, here the goal is radically different from reachability planning: the target behavior in our framework is not a specification of a goal state to reach but, rather, a description of a *process* one would like to be able to carry out *at runtime*. At every step there is possibly more than one action that can be carried out leaving the choice to the executor. Further, a target behavior may contain loops, which are typically ruled out in planning, similar to an IndiGolog program [27], i.e., a high-level procedure definable on top of a planning domain, for which one is typically interested to find an executable realization at runtime.

Each of the aforementioned systems uses additional features for selecting appropriate actions, e.g., FAtiMA focuses on a simple way of representing priorities among actions, and [28] requires that the choice of actions be intentional for NPCs. We stress that the aim here is not to achieve an *emergent believable behavior* for the NPC community according to the author's requirements; instead we aim at a *provably correct*

coordination strategy that would in fact work over any *infinite horizon* that fits in the specification of the target behavior. In particular compared then to ABL-like approaches [29] that are able to form joint goals for ensuring appropriate interaction of characters, our work is different in that (*i*) it decouples all "storyline" requirements from the behavior of NPCs into a target behavior for the entire system and (*ii*) it guarantees at design time whether it can be always enforced (and how) by means of the computed CG.

Finally, we borrow the notion of behavior composition from the literature, but not the typical representation of components as explicit-state transition systems. Here instead, we represent the environment and characters' behavior compactly through PDDL action specification. This gives rise to a complexity analysis that is different than the original one: It is actually then surprising that the problem remains EXPTIME-complete in spite of the exponentially smaller specification of the components. This is also reflected on the implementation for computing the composition in our system.

VIII. Conclusion

We have presented a system for modeling agents and devices in a virtual environment based on videogame technology enhanced with a high-level knowledge representation layer for enabling automated reasoning over the environment and the agents therein. In particular, we focused on behavior composition, however we see this approach as general in the sense that other forms of reasoning, planning, and automated synthesis can be handled in the same framework. Also, for now the space remains virtual in the game-world. Nonetheless, we envision that virtual environments can be attached to real smart spaces, becoming an intuitive and human-friendly way of controlling and visualizing real-world devices.

Acknowledgements. The authors acknowledge support of Sapienza Award 2013 "Spiritlets" project.

References

[1] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189–208, 1971.

[2] R. Reiter, *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001. [Online]. Available: http://www.cs.toronto.edu/cogrobo/kia/index.html

[3] M. Minsky, "A framework for representing knowledge," Cambridge, MA, USA, Tech. Rep., 1974.

[4] P. Bertoli, A. Cimatti, U. D. Lago, and M. Pistore. Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.5798

[5] H. L. S. Younes, M. L. Littman, D. Weissman, and J. Asmuth, "The first probabilistic track of the international planning competition," *J. Artif. Int. Res.*, vol. 24, no. 1, pp. 851–887, Dec. 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1622539

[6] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, "Automatic Composition of e-Services that Export their Behavior," in *Proc. of ICSOC 2003*, 2003, pp. 43–58.

[7] G. De Giacomo, F. Patrizi, and S. Sardiña, "Automatic behavior composition synthesis," *Artif. Intell.*, vol. 196, pp. 106–142, 2013.

[8] N. Yadav, P. Felli, G. De Giacomo, and S. Sardiña, "Supremal realizability of behaviors with uncontrollable exogenous events," in *Proc. of IJCAI 2013*, 2013.

[9] G. De Giacomo, M. Mecella, and F. Patrizi, "Automated service composition based on behaviors: The roman model," in *Web Services Foundations*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. Springer, 2014, pp. 189–214.

[10] S. Rabin, "Implementing a state machine language," in *AI Game Programming Wisdom*. Charles River Media, 2002, pp. 314–320.

[11] D. Isla, "Handling complexity in halo 2 AI," in *Game Developers Conference*, 2005.

[12] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins, "PDDL—The Planning Domain Definition Language," 1998. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212

[13] R. Milner, *Communication and Concurrency*, ser. Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., 1989.

[14] O. Kupferman, M. Y. Vardi, and P. Wolper, "An automata-theoretic approach to branching-time model checking," *Journal of the ACM*, vol. 47, no. 2, pp. 312–360, 2000.

[15] A. Muscholl and I. Walukiewicz, "A lower bound on web services composition," in *Proc. of FoSSaCS'07*, vol. 4423, 2007.

[16] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," *J. Comput. Syst. Sci.*, vol. 78, no. 3, pp. 911–938, 2012.

[17] H. Nishikawa, S. Yamamoto, M. Tamai, K. Nishigaki, T. Kitani, N. Shibata, K. Yasumoto, and M. Ito, "UbiREAL: Realistic Smartspace Simulator for Systematic Testing," in *Proc. Ubicom'06*, 2006.

[18] F. Araujo, M. Al-Zinati, J. Valente, D. Kuiper, and R. Zalila-Wenkstern, "Divas 4.0: A framework for the development of situated multi-agent based simulation systems," in *Proc. of AAMAS'13*, 2013.

[19] M. O. Riedl and V. Bulitko, "Interactive narrative: An intelligent systems approach," *AI Magazine*, vol. 34, no. 1, 2013.

[20] M. Mateas and A. Stern, "Towards integrating plot and character for interactive drama," in *In Working notes of the Social Intelligent Agents: The Human in the Loop Symposium. AAAI Fall Symposium Series. Menlo Park*, 2000, pp. 113–118. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3726

[21] J. Porteous and M. Cavazza, "Controlling narrative generation with planning trajectories: The role of constraints," in *Proc. ICIDS'09*, 2009, pp. 234–245.

[22] R. M. Young, "An overview of the mimesis architecture: Integrating intelligent narrative control into an existing gaming environment," in *Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 2001.

[23] H. Barber and D. Kudenko, "Generation of adaptive Dilemma-Based interactive narratives," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 309–326, 2009. [Online]. Available: http://dx.doi.org/10.1109/tciaig.2009.2037925

[24] R. Paul, D. Charles, M. McNeill, and D. McSherry, "MIST: An interactive storytelling system with variable character behavior," in *Interactive Storytelling*, 2010, vol. 6432, pp. 4–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16638-9_4

[25] R. M. Young, J. Thomas, C. Bevan, and B. A. Cassel, "Zócalo: A service-oriented architecture facilitating sharing of computational resources in interactive narrative research," in *Working Notes of the Workshop on Sharing Interactive Digital Storytelling Technologies at ICIDS'11*, 2011.

[26] J. Dias, S. Mascarenhas, and A. Paiva, "Fatima modular: Towards an agent architecture with a generic appraisal framework," in *International Workshop on Standards for Emotion Modeling*, 2011.

[27] G. De Giacomo, Y. Lespérance, H. J. Levesque, and S. Sardina, "IndiGolog: A High-Level programming language for embedded reasoning agents," in *Multi-Agent Programming: Languages, Tools and Applications*, 2009, pp. 31–72.

[28] M. O. Riedl and R. M. Young, "Narrative planning: balancing plot and character," *J. Artif. Int. Res.*, vol. 39, no. 1, pp. 217–268, Sep. 2010. [Online]. Available: http://www.cc.gatech.edu/~riedl/pubs/jair.pdf

[29] M. Mateas and A. Stern, "A behavior language: Joint action and behavioral idioms," in *Life-like Characters: Tools, Affective Functions and Applications*, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.106.1760