Toward an AI-based external scenario event controller for crisis response simulations

Dashley K. Rouwendal van Schijndel[*]
University of Oslo, Dept. of Technology Systems
d.k.rouwendal@its.uio.no

Audun Stolpe
Norwegian Computing Center, Dept. of Applied Research in Information Technology
audun.stolpe@nr.no

Jo E. Hannay
Norwegian Computing Center, Dept. of Applied Research in Information Technology
jo.hannay@nr.no

ABSTRACT

There is a need for tool support for structured planning, execution and analysis of simulation-based training for crisis response and management. As a central component of an architecture for such tool support, we outline the design of an AI-based scenario event controller. The event controller is a component that uses machine reasoning to compute the next state in a scenario, given the actions performed in the corresponding simulation (execution of the scenario). Scenarios are specified in Answer Set Programming (ASP), which is a logic programming language we use for automated planning of training scenarios. A plan encoding in ASP adds expressivity in scenario specification and enables machine reasoning. For exercise managers this gives AI-based tool support for before-action and during-action reviews to optimize learning. In line with Modelling and Simulation as as Service, our approach externalizes event control from any particular simulation platform. The scenario, and its unfolding in terms of events, is externalized as a service. This increases interoperability and enables scenarios to be designed and modified readily and rapidly to adapt to new training requirements.

Keywords

Scenario event controller, AI planning, Modelling and Simulation as a Service (MSaaS), simulation controller.

INTRODUCTION

There are two substantial challenges in simulation-based training for crisis response and management. One is the lingering pain over many years of insufficient planning and analysis of simulations and their intended learning effects (Hannay and Kikke 2019; Grunnan and Fridheim 2017). The other is the technological challenge of achieving the necessary interoprability between systems when constructing simulation-based training systems (Durlach 2018; Tolk 2012a; Edgren 2012). This article presents an approach where the events in a simulation are managed by a designated *simulation event controller*, which (1) supports machine reasoning in the unfolding of events of a scenario, and (2) is external and loosely coupled from any particular simulation platform.

The simulation event controller executes simulation scenarios specified in Answer Set Programming (ASP) (Lifschitz 2008); a declarative logic programming language, which supports automated reasoning of many forms—abductive, inductive and deductive. In this paper, we are using ASP in the so-called abductive mode for planning scenarios. A scenario is here understood as the sum total of ways of achieving a desired end or goal. The focus on goals (for example, put out the fire according to a given set of standardized procedures) relieves the scenario designer from specifying every single play that would fulfill those goals. The event controller's machine reasoning capability will instead generate all possible plays that meet the goals, step by step as the scenario unfolds through a particular play.

[*]corresponding author

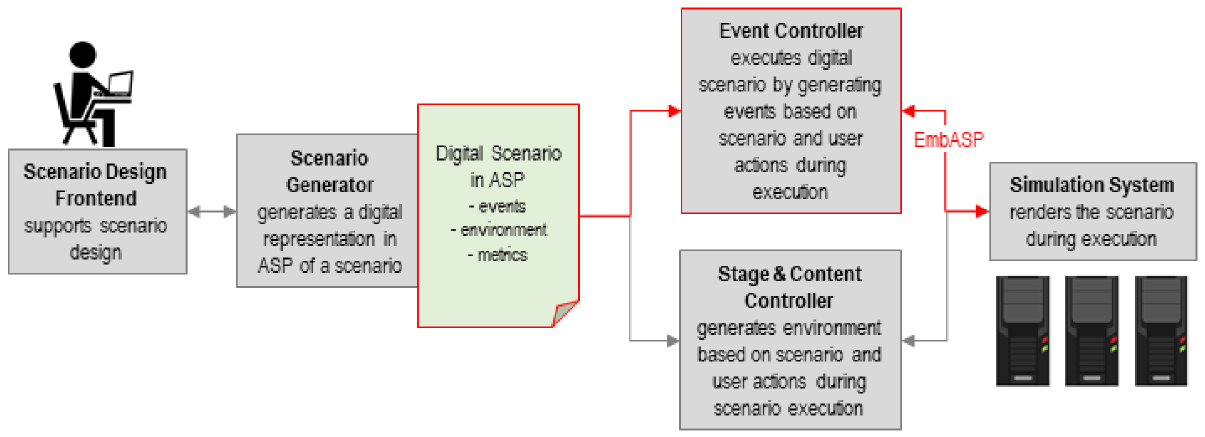

Scenario Design Frontend supports scenario design
Scenario Generator generates a digital representation in ASP of a scenario
Digital Scenario in ASP - events - environment - metrics
Event Controller executes digital scenario by generating events based on scenario and user actions during execution
EmbASP
Stage & Content Controller generates environment based on scenario and user actions during scenario execution
Simulation System renders the scenario during execution

Figure 1. Overall architecture of our approach, with the focus of this article indicated in red.

Compared to state of practice, this is a markedly different approach to both scenario specification and execution that should give scenario designers substantially enhanced means of expressivity and reasoning power at their disposal. The event controller can be used in "batch mode" in pre-action reviews of a scenario to examine and compare all possible (and relevant) plays of a scenario. This should increase the scenario designer's capability to design scenarios that are specifically aimed at stimulating goal-directed actions.

The loose coupling of the simulation event controller from any particular simulation platform is intended to decrease simulation construction time and increase interoperability and is in line with the Modelling and Simulation as a Service (MSaaS) paradigm (T. W. van den Berg et al. 2018; Hannay and T. W. van den Berg 2017). Currently, the construction of a simulation often consists of custom work in specific simulation platforms that requires months, and sometimes years, to complete. ASP enables an extremely concise and efficient specification of scenarios that lends itself readily to adaptations and modification. This flexibility should significantly reduce scenario (re-)design time, which is crucial for employing repetitions-based training according to learning principles. By externalizing simulation execution in our proposed event controller, construction time in the simulation platform is also reduced substantially; in fact, any specific simulation platform is, in our approach, simply a means of rendering whatever the event controller dictates. In MSaaS terms, both the digital scenario and the event controller provide services to be shared across simulation platforms and across instances of simulation-based training. These services facilitate interoperability at higher levels between systems (Tolk 2012b) by providing common data and functional resources.

In effect, the employment of machine reasoning and the externalizing of simulation execution in a single component is a step toward the automatic generation of simulations. This power comes from the expressivity of AI Planning and the event controller that is able to compute the appropriate event sequences for, in principle, any scenario. This picture is completed by an external *stage and content controller* (to be discussed at a later occasion) that generates the environment of a simulation. This methodology of AI-based scenario event control will give a significant contribution to dynamic scenario creation and control of execution which in term allows for part automation and dynamic customization control which allows Modelling and Simulation as a Service (MSaaS) as apposed to custom projects with extensive developments time and a lack of customization options. Figure 1 summarizes our approach.

The main body of this article elaborates our ideas laid out above in more detail. We contrast our approach to state of practice in specifying scenarios and scenario event control. We also remark on architecture.

ASP SCENARIO SPECIFICATIONS VERSUS STATE OF PRACTICE

There are standard methods for specifying scenarios in machine-readable format. The Coalition Battle Management Language (CBML) is a machine-readable language for expressing a commander's intent, across command and control (C2) systems, simulation systems and also autonomous systems (Simulation Interoperability Standards

Organization 2014; Simulation Interoperability Standards Organization 2019). The Military Scenario Definition Language (MSDL) SISO-STD-007-2008 (Simulation Interoperability Standards Organization 2008) is a standard for initializing or taking snapshots of the objects in simulations and their positions and states. Together, CBML and MSDL can be used to state what amounts to a machine-readable plan or digital scenario. Typically, CBML and MSDL are used to simulate operational plans in the defence domain; e.g, (Gautreau et al. 2014; Bruvoll et al. 2015). There are initiatives to define similar specification languages for domains other than the defence domain. The focus of CBML and MSDL is to specify the objects (also called entities) of a plan and then to specify what those objects should do, in the outset. This amounts to a script, or a static plan, that does not evolve as actual plays in a scenario unfold. There is no way to specify causal relationships between objects or to specify causal actions between objects, and therefore no inherent support for machine reasoning.

In contrast, planning in Answer Set Programming (ASP) is all about specifying causal relations. These causal relations describe how states in a scenario change subject to actions. To demonstrate this point we will be describing the method and implementation of a small scenario based on a kitchen fire. This example and accompanying description will be contrasted to current methodologies of creating traditional scrips for event and object control and the advantages an AI-based event controller would give. The method for encoding the event and state control using an ASP program will be demonstrated as to provide a clear picture of what this methodology would could look like when implemented.

To illustrate, consider the following mini scenario:

> A pan is sitting on a hot stove without a lid in an industrial kitchen. The pan contains cooking oil that will ignite and burn unless some measure is taken. In the best-case scenario, a trainee notices that the stove is turned on, recognizes the threat, and simply turns the stove off. Or, he may, sub-optimally, only realize the danger when the oil starts to give off smoke. Even at this point, there may still be time to prevent a fire if he acts quickly and both turns the stove off and puts a lid on the pan (thus countering that residual heat in the stove will ignite the oil even when the stove is turned off). Other things the trainee can do may be smart if performed at the right time but not otherwise, for instance turning on the fan to ventilate the smoke. If the stove is already turned off and the pan has a lid on it, then this is a useful thing to do. However, if there is still a chance of a fire in the pan then turning on the fan may cause the fire to spread to the ventilation system.

The corresponding ASP specification can be visualized as the transition diagram in Fig. 2. Here, the the edges are actions that the employee may perform, and the nodes are sets of states that are admissible by the program and the corresponding action.

For example, this ASP program has a causal relation

~on(stove) :- turnOff(stove)

stating (right to left) that turning off the stove causes the state of the stove to be “off” (or “not on”), and also the causal relation

heats(pan) :- ~turnOff(stove) and on(stove)

stating that not turning off the stove when the stove is on will heat the pan. By a technique known as *reification* causal rules are relativized to time points, or equivalently, to steps in the evolution of the scenario. Using the predicate "holds" to reify facts and "occurs" to reify actions the last rule becomes

holds(heats(pan), T+1) :- ~ occurs(turnOff(stove), T) and holds(on(stove), T)

The reified rule is a bit more verbose, due to the semantic ascent that is thereby effected. It asserts the truth of the proposition "the pan overheats" at time $T + 1$ if the "turn off stove" action does *not occur* at time T and "the stove is on" is true at the same time. A very important gain of reifying causal laws, is that in addition to modelling the effect of actions over time, it facilitates default reasoning about states and actions. For instance, all facts that are not affected by an action can be propagated forward in time in accordance with the classical *law of inertia*. That is, since facts tend to persist across time, only *changes* need to be computed. The effect of inertia is apparent in the nodes in Figure 2. Default reasoning can also be applied to actions as exemplified by the wait action in the same figure. It is supported by a default rule

occurs(wait, T) :- ~ occurs(A, T) and action(A)

saying that the wait action can be inferred to have occurred if there is no evidence that a positive action was performed.

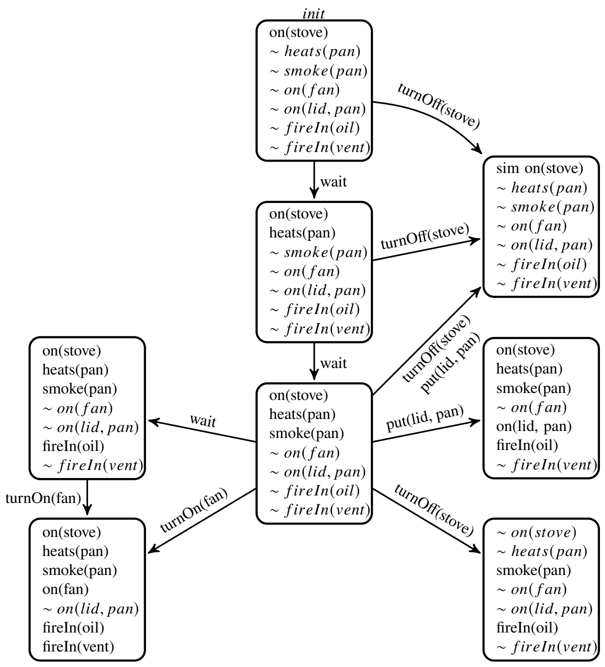


Figure 2. The kitchen scenario as a transition diagram

ASP-BASED EVENT CONTROLLER VERSUS STATE OF PRACTICE

Scenario specifications in MSDL and CBML are usually used to custom build simulations using some simulation platform, such as VR-Forces, VBS, Unity or Unreal. When constructing everything needed in a scenario in a game engine, two-dimensional (2D) or three-dimensional (3D) models that represent objects are programmed with scripts attached to them specifying their behavior, possible states and other attributes. These scripts dictate how an object reacts to actions in a simulation.

A training simulation for the industrial kitchen fire example above must include a pan filled with oil. The script attached to the pan object in Unity, say, would have a variable that keeps track of the current heat of the pan and a function that checks when the pan is over a certain amount of heat. When this is detected, another function in the script would be called to trigger the appropriate change of state. The overall scenario in terms of all the possible events would have to be programmed and stored in a separate script, waiting for triggers and actions to start the next part of the simulation for that object. In other words, all actions and objects' behavior would be programmed procedurally into the game engine; a process that is lengthy and time consuming for any simulation scenario of any relevant size.

In contrast, our approach replaces completely the need to program actions and the reactivity of objects into the simulation engine. The 2D or 3D models that represent objects still have to be programmed or picked from a library, but any scripting is, in our approach, simply for rendering visual effects, not for programming any logical functioning. In our approach, the meaning of actions and the resulting change of state are computed by the event controller.

The state change of objects in the scenario which is based on actions applied to objects, is calculated by the Clingo logic engine (Gebser et al. 2014). In a nutshell, using ASP as the driver of the application state of a simulation, means navigating a transition diagram such as that of Figure 2 when updating state in the simulation. This requires a two-way communication between the answer set solver and the rendering simulation platform.

In somewhat more detail, the output of an ASP program is a set of states or *models* (in the logical sense) that satisfy the program in question. For the event controller, we are exploiting this feature to evolve a simulation in a stepwise manner, setting up a correspondence between the state of the simulation and a model of the associated ASP program. To that end, the associated ASP programs are formulated as *causal theories* (refs); that is, as set of causal relationships within an individual system or domain. We are particularly interested in *actions*, whence a causal theory, in the more specific sense (following refs) that that term is used in the present paper, is a description of the changes caused by executing actions that produces a sequence of models. This sequence represents the world as it unfolds over time corresponding to the choice of one or more agents. Due to the nature of ASP causal theories, this

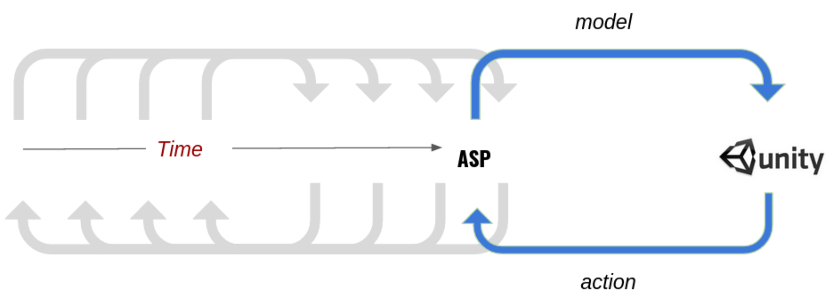


Figure 3. The ASP-Unity loop.

communication follows a very simple and easy-to-implement protocol illustrated in Figure3 for Unity simulation platform.[1]

This architecture reduces the responsibility of the simulation or game engine to that of rendering the scene and recording actions. Once an action is recorded, a message is sent to ASP which computes the consequences of that action according to the causal theory. The output is a complete description, in the form of a model, of the state of *all* objects in the simulation scenario. This includes the state of the objects that are affected by that action, as well as the state of objects that are not, the state of the latter being propagated forward in time by default.

Now, the usual way to handle events in a 3D engine like Unity is to associate actions with *callback functions* in a mediator object that manages the interaction between listening objects. Each callback function is bound to a *single* object and implements the effect of the action in question on *that object*. In order to update the state of the entire scenario or simulation, therefore, each callback listening to that action must be executed to yield the new state of the involved objects. Moreover these new object states may interact among themselves and trigger ripple effects, as for instance when a fan is turned on and there's a fire in the pan sitting under it.

Each of these ripple effects require their own callbacks. Hence, the number of callbacks required will in general grow quickly with the complexity of the scene. In contrast, when ASP is made to act as the driver of the application state, one can always make do with only the function that sets up the ASP-Game Engine communication protocol. All effects of actions, all indirect effects of action, and every interaction between objects is computed by ASP and recorded in a single returned model. At the face of it therefore, not even considering the benefits of symbolic AI for reasoning about dynamic domains, there is a major architectural simplification to be gained from factoring the event logic out of the game engine itself.

It is key to understand that we are replacing the combination of object script's hard coded behavioral functions, callbacks and scenario scripts waiting for triggers, with a single centralized AI engine that takes over these duties by taking a performed action and creating a model filled with all the new object states, which is entirely determined by a causal theory.

A major advantage of using an AI language such as ASP and its corresponding AI engine (Clingo) to control the flow of the scenario execution and object state is that one does not have to reprogram large amounts of behavioral scripts for objects or the overall scenario script. By simply changing the causal relations one can dramatically change the behavior of objects in response to actions on them. In a traditional implementation of the example of the industrial kitchen, the pan may be programmed to hold specific functionality. Perhaps the pan can be releasing smoke or on fire. If one wanted to create a completely new type of state or situation the pan can be in, one would have to go into the pan script and manually code in that functionality. If one wanted a pan on fire to unlock all the doors in the building (as a fire safety mechanism), one would need to reprogram the script for the pans, doors or the manager object that checks for this. In our apporach, it suffices to add a causal rule to the ASP rule file that reads

```
holds(unlock(door), T):- occurs(onFire(pan), T-1)
```

This one rule states that if the pan is on fire, all objects that are a 'door' become unlocked at the next time step of the program.

[1]For the logically inclined, the communication loop in Figure 3 can be considered a function from actions to the power set of facts in the Herbrand base of the causal theory. That is, the loop expresses a function from action expressions to the set of all ground atoms that can be formed from predicate symbols in the causal theory by substituting symbolic constants, i.e. names, for variables. The thing to note about this function is that its return value consists of *multiple facts*, each one recording the state of an object at the same singular point in time. Intuitively this return value is a *time slice* or *snapshot* of the entire system at that particular time.

This enables very quick adaption of a scenario without having to spend time-intensive changes to procedural scripts within the simulation/game engine. The additional future benefit is that when scoring systems are integrated in a simulation system to identify points of attention for which the trainee needs more training, scenarios can be very easily adapted with minimal coding and changes. This allows the trainee to do iterative training while the scenario adapts to their needs. This could even in part be done automatically by having a set of adaptive causal rules that are activated in the relevant circumstances. This is actually a fairly low-hanging fruit since adaptiveness is just another name for default reasoning in this context: ways of adapting is a matter of keeping tabs on the exceptions that defeat certain rules and reinstate others.

EXAMPLE OF AN EVENT CONTROLLER IMPLEMENTATION.

In its most basic form, we illustrate our approach by outlining an implementation of an ASP-based event controller in conjunction with Unity, which is a video game engine.

The Unity engine communicates with the reasoning engine via embASP (Calimeri, Fuscà, et al. 2019). embASP is "A framework for the integration (embedding) of Logic Programming in external systems for generic applications. It helps developers at designing and implementing complex reasoning tasks by means of solvers on different platforms." This is an example of an event controller implementation. The event controller is a system that controls the flow of the training scenario. To integrate this with the Unity engine we will be using the embASP $C^{\#}$ implementation.

In other words, this framework integrates ASP in other programming environments that use popular object oriented programming languages such $C^{\#}$.

There is an example where embASP is implemented into Unity to utilize ASP for the purpose of AI decision making (Calimeri, Germano, et al. 2018). This paper uses embASP to implement a decision making process for an AI character. It takes the the game "Pacman" and makes Pacman an AI character rather then player controlled, driven by ASP. The paper by Calimeri et al described the actions of Pacman based on a threshold calculation fed by the position of a set number of variables in the scene (location of pellets, location of enemy ghost, location/using powerup) and then choosing from a limited set of options for AI decision making (picking a direction to move).

But in distinction with Calamari et al we are not using the embASP and numeric calculations for AI decision making of an actor, but for global scenario event control via actions (state changes) on scenario relevant objects, which in turn drives the scenario forward via the causal theory. This paper utilizes embASP for scenario event handling across the entire scene based upon the actions of a user. This paper proposes to use the ASP reasoning ability to handle all aspects of state changes, thereby controlling the entire simulation via a causal theory, while still allowing a user to be the cause of the change by conducting an action. This changes the implementation from a game character AI engine to that of a scenario event control system.

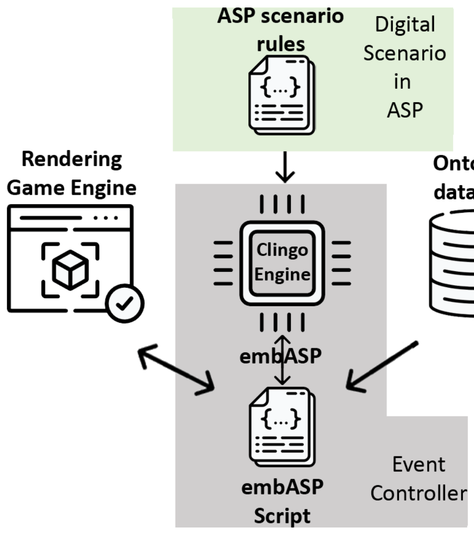


Figure 4. Implementation Overview.

The overview of the event manager architecture as shown in Figure 4, consists of a GameObject with an attached script, in the the embASP specific format. This would include the objects possible states and other object specific information. A GameObject in unity is defined as "the fundamental objects in Unity that represent characters, props and scenery. They do not accomplish much in themselves but they act as containers for Components, which

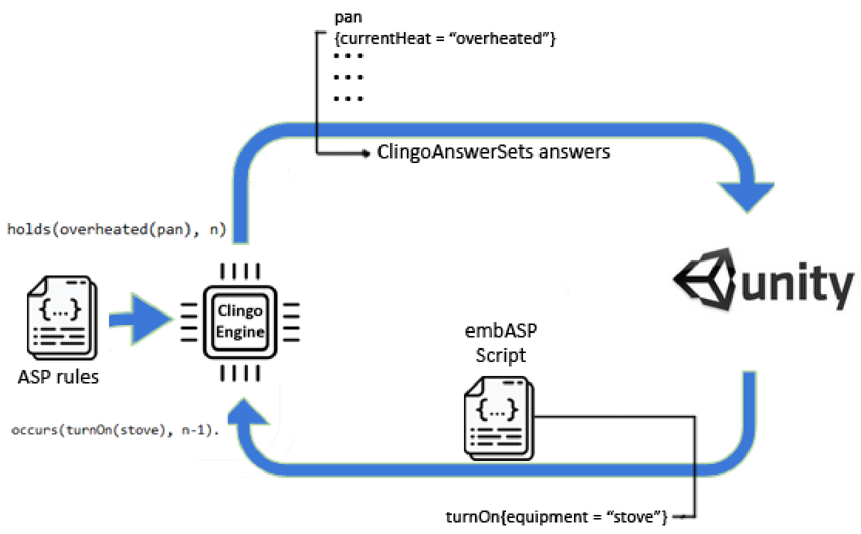

pan
{currentHeat = "overheated"}
...
ClingoAnswerSets answers
holds(overheated(pan), n)
Clingo Engine
ASP rules
unity
embASP Script
occurs(turnOn(stove), n-1).
turnOn{equipment = "stove"}

Figure 5. The ASP-Unity detailed loop.

implement the real functionality."[2] In something like the kitchen scenario, a gameObject would be the furniture, building, lighting, avatars and essentially everything else in the scene. The specific information that describes what it can do is defined in the attached script. In the kitchen fire training scene the ASP relevant objects would be the objects onto which actions need to be performed to complete the training exercises, such as the pan, stove, ventilation, etc. A ceiling lamp would be an example of a unity GameObject, but one that does not require the functionality of a embASP Script since it requires no change of state for the completion of the training scenario.

This would mean that those specific gameObjects that are required to be acted on to complete the training scenario, need their scrips to be formatted in accordance to the embASP $C^{\#}$ requirements so that it can be interpreted by the AI engine. Figure 6 has a simple example of what this would look like. This would correspond to some simple changes in how a class would be setup compared to the traditional unity scripting setup, so that it can be interpreted by embASP. This format is defined in the $C^{\#}$ implementation for embASP.[3] The ontology database as show in figure 4 contains the descriptions and necessary features required by various objects that are to be simulated in a scene. This information is partially what will be required to build embASP scripts for gameObjects. The information in this ontology would need to be formatted into the $C^{\#}$ embASP script format.

The state change of objects in the scenario which is based on actions applied to objects, is calculated by the Clingo logic engine (Gebser et al. 2014). Figure 5 shows the ASP-Unity loop with the actions from unity being passed to the ASP reasoning engine (which utilizes Clingo) and it passing back a "model" which describes the meaning of the actions and which is what determines the new "state" of objects within the Unity scene.

A benefit of using an ASP based reasoning engine is that it keeps the information that was previously communicated to it when the scene was created, retaining the previous state of all the objects in the scene. The ASP engine only requires the actions that are applied to the ASP gameObjects. This has the benefit that the object state does not have to be re-iterated to the ASP engine. Any computational complexity that requires resources would be minimized to a limited amount of calls to the ASP engine, which happens only when a change state occurs in a relevant object.

The added explicitly to the scenario event management leaves no doubt as to the determined effects of actions within a training simulation. Even when causal effects are not expressed by a rule within the ASP rule file, the ability of default reasoning which is integrated in ASP allows for the creation of models based on actions without having to specifically define in detail the causal effect of every action. This allows for assigning default values to constraint variables or to leave them undefined while still being able to create a model which will describe the change of state to ASP objects within the simulation.

[2]https://docs.unity3d.com/560/Documentation/Manual/class-GameObject.html
[3]https://www.mat.unical.it/calimeri/projects/embasp/#License

The ASP scenario rule file contains the information that creates the simulation's scenario structure. It holds the generated scenario, but in a series of rules that depending on the actions will change the state of objects in a scene. This allows for the entire scene to be played out simply with actions(object state changes). In the kitchen fire example, if the pan is heating up at a dangerous rate, there are a number of actions a trainee could take. 1. switch off the heat source, 2. switch on the ventilation, 3. take the pan off the heat source. The ASP scenario rule file would hold the information that defines the change of state of objects in the scene based on these 3 actions. Action 1 and 3 may end the scenario successfully, while action 2 would create airflow, which would lead to a fire and the scenario continuing from there. This example is simplified, but the idea of action and state change increases exponentially and the ASP reasoning engine is designed to handle this kind of exponential growth. EmbASP Utilizes the defined ASP rules as the driver of a scenario. An example of such a rule can be seen below

holds(heats(pan), T) :- occurs(turnOn(stove), T-1).

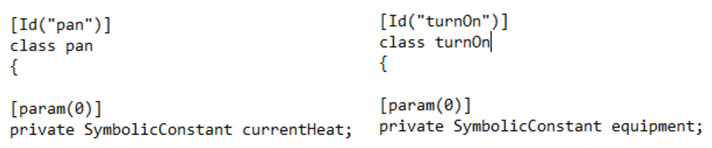
In the example of the kitchen fire, the stove switch, having changed from the off to the on position would be registered as a change of state in the simulation and send as an action to the ASP reasoning engine. The ASP reasoning engine would apply this action to its knowledge base and compare it to the rule specified above. In this case the rule would detect that the pan will overheat in the next time step 'T' if the stove has been switched to the turnOn position. The ASP reasoning engine would return the information that the pan is "overheated" in the next time step and this would be applied to the pan's state in the embASP script in Unity. This would complete a simple scenario event as controlled by the ASP reasoning engine as portrayed in figure 5.

The ASP reasoning engine can be externalized from multiple simulation engines to function as a central point for calculating cause and effect from various simulation sources. The generated models would be returned to each simulation and the state updates could be implemented in each simulation accordingly. These models hold the state information based upon the latest action in accordance to the placement in time (time step) of all the scenario relevant object (ASP objects) in the simulation scene. The time step is the factor that is synchronised in all simulations and kept track of in the centralized system. This allows the ASP driven scenario event controller to return models of all causes and effects. There would be no need to register objects in such a manner as to create forms of ignorance between objects to save on computational complexity. ASP is created to handle large amounts of interdependent data and with the rule checks only conducted at the time of an action (change of state) within the simulation, the amount of times the ASP engine needs to be called is reduced as to minimize the amount of computations necessary.

A NOTE ON SIMULATION ARCHITECTURE

The fact that our approach centralized event control completely requires some remarks in relation to state of practice that often favors distributed and delegated approaches. Figure 7 shows the layout of a simulation according to the High Level Architecture (HLA) (Simulation Interoperability Standards Organization 2016) standard for distributed simulation systems (Petty and Gustavson 2012; Kuhl et al. 1999). In HLA, the simulation software that make up a simulation system are distributed into modules called *federates*. Federates are be combined to form a *federation*, coordinated by a *runtime infrastructure* (RTI). HLA prescribes a publish/subscribe protocol: federates publish object attributes and interactions between attributes, and federates may subscribe to updates of published attributes and interactions. The RTI coordinates these messages. Federates may also query the RTI on-the-spot for updates. HLA thus adheres to the *Observer* and *Mediator* patterns (Gamma et al. 1995). This architecture enables participants to train on various simulation platforms distributed geographically, but it is also common to use HLA for non-distributed simulations with only a single federate.

The RTI administers the shared state of the federation and also administers time according to several time management schemes (Fujimoto 2000). The federates themselves are ignorant with respect to the shared state; i.e., a federate is not aware of the states of other federates, except for the parts of the shared state that the federate



```
[Id("pan")]
class pan
{

[param(0)]
private SymbolicConstant currentHeat;
```

```
[Id("turnOn")]
class turnOn
{

[param(0)]
private SymbolicConstant equipment;
```

Figure 6. embASP gameObject Script.

subscribes to. For example, a federate may simulate an object that deposits fire extinguishing measures toward an object that represents a fire at a specified location without needing to know what other objects are in the simulation. Nevertheless, a federate who wishes to monitor if any of its objects are affected by the fire extinguishing measures can subscribe to the part of the shared state that reflects the relevant information.

Thus, in HLA, simulation control resides decoupled in the federates. This effectively separates concerns, which is necessary, given the intricate procedural programming that is necessary to implement simulations. The distributed and delegated concept is taken even further in edge computing approaches such as SpatialOS.[4]

In our approach, scenario control, and thus simulation control, is centralized and not delegated in any way. What is delegated, is the particular rendering mechanisms of simulation and game engines. There are several points to be made on this.

First, the machine reasoning that our approach employs to maintain a focus on goal achievement requires a component that not only holds the shared state of the system (as does the RTI in some sense), but is also able to compute future plays and states of a scenario. In the outset, it is more efficient to do this centrally.

Secondly, the need for separation of concerns by distribution and delegation that is necessitated by procedural programming of object behavior is not present with declarative programming in ASP.

Third, distributed training can still take place with our approach. In fact, this is made even simpler by the fact that the particular simulation and game engines that trainees use during training simply render the unfolding of a centrally computed scenario. As mentioned above, rapid changes and adaptations can be applied even during training in the ASP scenario specification, with no need to reprogram each simulation and game engine.

These benefits and, indeed the architectural choices in our approach are very much in line with the MSaaS reference architecture (Hannay and T. W. van den Berg 2017). Our external scenario event controller corresponds to the *simulation controller service* in (Hannay, T. van den Berg, et al. 2020).

In HLA, the objects and interactions that are shared (i.e., whose attributes are published and subscribed) among federates in a federation are declared in a *Federation Object Model* (FOM), which is input to the RTI. Thus, all federates must relate to this data declaration during runtime, and also during design time. At run-time, the declarations in the FOM give rise to variables that constitute the shared state of the federation.

FOMs are relevant to our approach, since they are, in effect, ontologies of the relevant objects and their possible interactions for a particular domain. A FOM or an ontology functions as a library of standardized objects and interactions in the form of causal relations that can be imported into a particular scenario, prior to specifying the particulars of that scenario.

DISCUSSION

So why is using an AI reasoning engine an advantage as opposed to simply changing the existing scripts already attached to objects in the simulation? We mentioned earlier that there is a major architectural simplification to be gained from factoring the event logic out of the game engine itself because all actions are returned in a single model.

[4]https://www.improbable.io/blog/how-spatialos-works-with-game-engines

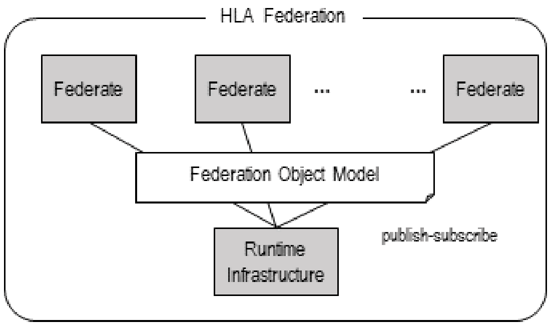


Figure 7. The High-Level-Architecture (Simulation Interoperability Standards Organization 2016) for federating distributed simulation systems.

Rather then having a distributed system where a possible action can cause a chain reaction between objects, each having to perform a sub-routine, the entire chain of change states would be calculated in a single cycle via the AI reasoning engine.

Another strengths is that you can edit a training scenario without having to touch the rendering engine (game engine) or scripts. You can drastically change the way a scenario is played out by changing a minimal number of rules using the ASP causal relations. This allows for both rapid prototyping and simulation re-use, where a single created scenario could be utilised over and over to train different skills, procedures and goals.

As part of the ExManSim architecture, the consistent machine readable format of generating ASP based rule-sets can in part be automated from information specifications in the design phase of the simulation. When a user defines specifications withing a digital format such as a web interface, these specifications can be added as simple rules into an ASP rule file. If the user designing the scenario adds more information that would alter the causal effects from previously defined rules, the ASP engine would be able to deal with this quite simply because it is designed to analyze the rule-set and create a solution model based on the resulting outcome of all the rules. For instance, if two rules were to cancel each other out, it would know to ignore both these rules all together and not use them when calculating a solution.

An additional benefit of a externalized event controller based on ASP is that it could be moved away from the client side software and centrally managed on an external server, this allows for synchronized updates of the simulation to multiple clients and allows for greater control. Competitive multiplayer games for instance, don't usually have client side hit detection because of the prevalence of cheating, instead hit detection is handled on server side. In the simulation using the ASP based event manager you could do this for every action of the simulation state and the returning information would define the state of every object in the scene. This is sort of how a multiplayer game currently works because every players game needs to be in sync with all the others. An external ASP based handler could simplify this because all effects of actions, all indirect effects of action, and every interaction between objects is computed by ASP and recorded in a single returned model in accordance with the systems time step.

Because such an AI engine can also calculate the optimum solution to the set training scenarios based on its rule set there is a potential for future research and products which allow for scoring mechanisms which could collect information from the trainee performance and determine the optimum training focus that a trainee would require which allows for quick adaption of the scenario accordingly. It would not require rebuilding or changing of the scenario but instead, small changes in the causal theory. To return to the kitchen example, the previous runs of the simulation could potentially find that the trainee handles the situation perfectly before a fire happens, but after the fire starts, the trainee does not take the appropriate actions. The ASP rules can be adjusted quite easily to make sure that a fire always occurs or that a different type of fire occurs, so that the areas where the trainee needs to most training become the new focus of the scenario. The rapid customization of the AI-based external scenario event handler allows for this flexibility and specification without having to reprogram the scenario from scratch.

CONCLUSION

This paper has argued the benefits of using Answer Set Programming as a simulation event controller, backed by known benefits of ASP and new ones deriving from its contextual use as a scenario event controller. We created a communication loop which allows for the change of state in ASP simulation objects based upon the actions performed in the simulation by a trainee and the ASP rule set. This information is interpreted by the Glingo engine and returns a model of state changes which is interpreted by the simulation engine and applied to the corresponding objects.

There are a large number of benefits that an ASP based externalized AI engine for event control offer, which is why we propose this as a new methodology which should be applied to automated simulation systems such as the ExManSim project. The benefits include fast prototyping, dynamic changing of scenario's, high specificity with ability fore default reasoning and a centralized infrastructure for calculating the simulations world state. This coupled with the future potential of creating automatic customized scenario creation based on the trainees needs from iterative training sessions makes it a particularly powerful methodology for controlling scenario events.

Acknowledgements

This research is funded by the Research Council of Norway under project no. 282081 "MixStrEx".

REFERENCES

Bruvoll, S., Hannay, J. E., Svendsen, G. K., Asprusten, M. L., Fauske, K. M., Kvernelv, V. B., Løvlid, R. A., and Hyndøy, J. I. (2015). "Simulation-Supported Wargaming for Analysis of Plans". In: *Proc. NATO Modelling and Simulation Group Symp. on M&S Support to Operational Tasks Including War Gaming, Logistics, Cyber Defence (STO-MP-MSG-133)*.

Calimeri, F., Fuscà, D., and Germano, S. (2019). "Fostering the Use of Declarative Formalisms for Real-World Applications: The EmbASP Framework". In: *New Generation Computing* 37.1.

Calimeri, F., Germano, S., Ianni, G., Pacenza, F., Perri, S., and Zangari, J. (2018). "Integrating Rule-Based AI Tools into Mainstream Game Development". In: *Lecture Notes in Computer Science* 11092.

Durlach, P. J. (2018). "Can we talk? Semantic Interoperability and the Synthetic Training Environment". In: *Proc. Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2006*. National Training and Simulation Association.

Edgren, M. G. (2012). "Cloud-Enabled Modular Services: A Framework for Cost-Effective Collaboration". In: *Proc. NATO Modelling and Simulation Group Symp. on Transforming Defence through Modelling and Simulation—Opportunities and Challenges (STO-MP-MSG-094)*.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*. Wiley-Interscience.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gautreau, B., Khimeche, L., De Reus, N., Heffner, K., and Mevassvik, O. M. (2014). "A Proposed Engineering Process and Prototype Toolset for Developing C2-to-Simulation Interoperability Solutions". In: *19th Int'l Command and Control Research and Technology Symposium (ICCRTS)*.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2014). "Clingo = ASP + Control: Preliminary Report, 2014". In: *Preliminary Report*.

Grunnan, T. and Fridheim, H. (2017). "Planning and conducting crisis management exercises for decision-making: the do's and don'ts". In: *EURO Journal on Decision Processes* 5, pp. 79–95.

Hannay, J. E. and Kikke, Y. (2019). "Structured crisis training with mixed reality simulations". In: *Proc. 16th Int'l Conf. Information Systems for Crisis Response and Management (ISCRAM)*, pp. 1310–1319.

Hannay, J. E., van den Berg, T., Gallant, S., and Gupton, K. (2020). "Modeling and Simulation as a Service infrastructure capabilities for discovery, composition and execution of simulation services." In: *J. Defense Modeling and Simulation: Applications, Methodology, Technology*, pp. 5–28.

Hannay, J. E. and van den Berg, T. W. (2017). "The NATO MSG-136 Reference Architecture for M&S as a Service". In: *Proc. NATO Modelling and Simulation Group Symp. on M&S Technologies and Standards for Enabling Alliance Interoperability and Pervasive M&S Applications (STO-MP-MSG-149)*. Lisbon, Portugal: NATO Science and Technology Organization.

Kuhl, F., Weatherly, R., and Dahmann, J. (1999). *Creating Computer Simulations—An Introduction to the High Level Architecture*. Prentice Hall PTR.

Lifschitz, V. (2008). "What is Answer Set Programming?" In: *Proc. 23rd National Conference on Artificial Intelligence (AAAI'08) – Volume 3*. AAAI Press, pp. 1594–1597.

Petty, M. D. and Gustavson, P. (2012). "Combat Modeling with the High Level Architecture and Base Object Models". In: *Engineering Principles of Combat Modeling and Distributed Simulation*. Ed. by A. Tolk. Wiley. Chap. 19, pp. 413–448.

Simulation Interoperability Standards Organization (2008). *SISO-STD-007-2008 – Standard for Military Scenario Definition Language (MSDL)*.

Simulation Interoperability Standards Organization (2014). *SISO-STD-011-2014 – Standard for Coalition Battle Management Language (C-BML) Phase 1, Version 1.0*.

Simulation Interoperability Standards Organization (2016). *SISO-PN-016-2016 – Product Nomination for High-Level Architecture Version 3.0*.

Simulation Interoperability Standards Organization (2019). *The Command and Control Systems – Simulation Systems Interoperation (C2SIM) Product Development Group (PDG) and Product Support Group (PSG)*.

Tolk, A. (2012a). “Integration of M&S Solutions into the Operational Environment”. In: *Engineering Principles of Combat Modeling and Distributed Simulation*. Ed. by A. Tolk. Hoboken (NJ), USA: John Wiley & Sons. Chap. 15, pp. 295–327.

Tolk, A. (2012b). “Standards for Distributed Simulation”. In: *Engineering Principles of Combat Modeling and Distributed Simulation*. Ed. by A. Tolk. Hoboken (NJ), USA: John Wiley & Sons. Chap. 12, pp. 209–241.

van den Berg, T. W., Huiskamp, W., Siegfried, R., Lloyd, J., Grom, A., and Phillips, R. (2018). “Modelling and Simulation as a Service: Rapid deployment of interoperable and credible simulation environments – an overview of NATO MSG-136”. In: *Proc. 2018 Winter Simulation Innovation Workshop*. 18W-SIW-018.