Simulation
XX(X):1–34

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

A logic-based event controller for means-end reasoning in simulation environments

Audun Stolpe[1], Ivar Rummelhoff[1] and Jo Erskine Hannay[2]

Abstract

Simulation games are designed to cultivate expertise and rehearse particular skill sets. In order to yield longitudinal effects, sequences of events must be crafted to yield intended learning outcomes, sometimes by focusing on particularly difficult situations and replaying variants. The present paper develops a logic-based approach for encoding the interrelation between action, events and objects in a manner that allows the resulting scenario description to immediately be executed in a game development environment. This has the dual effect of decoupling the description of a scenario from the simulation platform itself, as well as supporting iterative and flexible development of learning content. To this end, we provide three interrelated components: First, we develop a scenario description language based on Answer Set Programming. The language is designed to allow an automated reasoner to deduce a schedule of the future events that are caused by an action taken in a given simulation environment. Secondly, we define a protocol for exchanging actions and computed futures between, respectively, the simulation environment and the external automated reasoner. Finally, as a proof of concept, we develop an API for the Unity Real-Time Development Platform that implements the protocol and offers a software framework for for connecting the computed future events to concrete game objects. This allows the game to evolve coherently from the specification. We argue that the resulting system inherits capabilities for artificial commonsense reasoning from its declarative basis which are useful for reasoning about an evolving emergency incident or training scenario.

Keywords

Simulation event controller, Scenario specification, Automated reasoning, Serious gaming.

1 Introduction

Serious games for emergency preparedness are designed to cultivate and nurture the areas of expertise that are required for efficiently responding to incidents and crises. Examples include games for medical training, information security and cyber defence, and, more generally, for civilian and military exigencies. Such games typically target skills in mission planning,

[1]Norwegian Computing Center, Norway
[2]Center for Effective Digitalization of the Public Sector, Simula Metropolitan, Oslo, Norway

Corresponding author:
Audun Stolpe, The Norwegian Computing Center, Pb. 114 Blindern, NO-0314, Norway. Phone: +47 22852554.
Email: audun.stolpe@nr.no

operations support and analysis. Common to all, they are designed to simulate real-world processes and protocols, usually with a carefully regimented story in mind.

In order to guide the unfolding of the story towards its intended purposes and desired learning objectives, a serious game should be based on the concept of *deliberate practice*.[1] That is, it should offer targeted and task-centered training based on instructions, including sequences of events crafted to enhance personalized learning – e.g. by focusing on particularly difficult situations, by immediate replays of situations and by immediate constructive feedback from the environment. That this is non-trivial is perhaps witnessed by the fact that simulation-based training often lacks precisely this form of deliberate scenario design.[2]

What is called for is a global view of a simulation scenario and its *inherent narrative possibilities* broken down into its *causal chains* and the *courses of action* that are supported by the corresponding *means-end relationships*.[3] The specification of the simulation scenario should, moreover, be *executable*, but at the same time sufficiently *abstract and conceptual* to move the domain-specific aspects of implementing the scenario from the programmer towards the workflow of the subject-matter expert.

Model-driven frameworks have been proposed that fit part of this bill. In particular, they usually offer an abstract view of simulations, focusing effort on the conceptual representation of the knowledge and activities that govern a particular application domain. The most widely disseminated and studied example of this is the Discrete Event System Specification (DEVS) of Zeigler et al.[4] which is based on automata theory with mixed-in features from process modelling and object orientation. In DEVS a simulation is modelled as either a discrete event system described by a state transition table, a continuous state system, described by as set of differential equations, or as a mixture of the two.

Without disputing the merits of DEVS, or of model driven development more generally, the present paper wishes to draw attention to the relative advantages of a logic-based approach. On the logic-based approach, a simulation scenario is a *calculus of actions and events* that describe the effects of actions and the interaction between objects in space and time, and hence implicitly all possible evolutions. In line with established practice we shall refer to such a calculus as an *action description* or a *causal theory*.[5–9] The two terms will be used interchangeably to emphasise different aspects – the *agentive* and the *causal* respectively – depending on the surrounding discussion.

The principal advantages of the logic-based approach can be summed up in four points: First, event calculi/causal theories are rooted in a long tradition in declarative artificial intelligence known as *default reasoning*[10,11] or *qualitative uncertain inference*.[12] Default reasoning is the process of drawing conclusions on the basis of what is true in typical cases, thereby also allowing inferences to be altered or retracted should the situation turn into to an exceptional one. This is a pattern that fits well with reasoning about cause and effect in general, and thus with the evolution of an emergency incident in particular. For example, in a workplace emergency an incident commander would typically evacuate people through the designated emergency exits. But not in the event, say, that the doors remain locked or that the exit access is blocked by fire or smoke. More examples are provided in Section 9.

Secondly, we will show that an interesting and useful range of such causal rules can be expressed in a simple, *fully declarative* rule language. Declarativeness empowers the scenario designer to focus exclusively on describing *what happens when* without concerning himself with *how* those events are to be executed. Accordingly, scenario design becomes neither a software development nor a systems engineering problem, but rather a specification task: write rules not processes.* Moreover, this property aligns the logic-based approach with model-driven approaches in that the model/specification becomes an independent asset decoupled from the simulation environment itself. The conceptual issues involved in designing a useful training scenario are separated from the programming logic required to execute it.

*Granted, domain experts should perhaps not be burdened with knowing how to express themselves directly in the syntax we are proposing, so the underlying assumption here is that they will be presented with some sort of user-friendly interface to assists in composing complex rules from simple building blocks. A candidate would be BlawX,† a web-based, visual development environment based on Google's Blockly. Our action description language can easily be implemented in BlawX.

Thirdly, when cast as a causal theory, in the current technical sense of the term, a scenario description becomes a mathematically well-defined object – ultimately a formula of first-order logic – itself amenable to logical analysis. Thus, a number of interesting mathematical questions arise already in the meta-theory *about* the causal theory. An example is the precise conditions under which a causal theory predicts a *unique* future of events as the causal consequences of a single action. As explained in Section 8 this is a necessary condition for a simulation scenario to evolve predictably as per the decisions of the designer.

With the aim of leveraging these advantages into modelling and simulation, the present paper develops an approach designed to employ a logical description of a training scenario as an *executable specification* that drives the evolution of a simulation scenario. The idea is to use an off-the-shelf automated reasoning engine as an *external event controller* by having the simulation environment call back to it and query the scenario specification for the *default future of events* given the actions that have been performed in the simulation.

We carry out this program in three steps, that also represent the novel contributions of this paper: first, we develop an action description language called $\mathcal{ALID}$. It is implemented as domain specific language based on *answer set programming*,[6,8,13] a declarative programming language that supports default reasoning. $\mathcal{ALID}$ is a generalization of the simpler language $\mathcal{AL}$[5,6] that introduces two new features: first, $\mathcal{ALID}$ has *durative* actions and events. Secondly, $\mathcal{ALID}$ introduces a concept of *inferred time*, meaning that the timeline of the default future is *derived* from the time indices of the actions that have been performed in the simulation environment. This is essential for aligning the computed schedule with the simulation's timeline, ensuring that events scheduled for execution in the simulation environment are plotted on the simulation's clock.

In the second step, we define a stateless protocol for exchanging information between a reasoning engine and a simulation platform. This protocol is quite abstract, and can act as an interface between any system that emits actions and consumes a schedule of future events, on the one hand, and a system that consumes actions and emits a schedule on the other.

In the third step we implement the proposed protocol in a piece of software we refer to as the ScenarioEngine simulation event controller. It was developed as a proof of concept specifically for the Clingo[‡] answer set solver and the Unity Real-Time Development Platform.[§]

The present paper is meant to be a foundational study. Many important practical topics are put to one side for now. For instance, we only consider single-player scenarios, and we do not consider optimizations such as asynchronous execution of time-consuming reasoning tasks. Also, empirical user-facing studies are postponed to future work.

The paper is organized as follows: Section 2 gives an overview of related work. Section 3 offers a brief introduction to answer set programming and the stable model semantics. In Section 4, the general concept of a causal theory is explained, and a running example is introduced. Section 5 presents the, for our work seminal, action description language $\mathcal{AL}$, and argues that it is not sufficiently expressive for our purposes. Requirements for a more expressive language are formulated in Section 6 through the description of a stateless protocol for communication between a reasoning engine and a simulation platform. This leads to the definition of the language $\mathcal{ALID}$ in Section 7. Section 8 contains a brief statement and discussion of the aforementioned property of determinism. A rigorous formal proof is relegated to the appendix. Turning to more practical matters, we elaborate on the running example in Section 9 as an illustration of how to use $\mathcal{ALID}$ to represent and reason about the different events and the possible lines of evolution that a scenario presents. In Section 10 we describe a $C^{\#}$ implementation of the protocol from Section 6 for Unity and the answer set solver Clingo. We take stock and discuss some possible lines of further research in Section 11.

2 Related work

The present research develops a logic-based approach for specifying and controlling the overall causality and means-end reasoning of a simulation scenario. Several

[‡] https://potassco.org/clingo/
[§] `https://unity.com/`

other approaches relate to our work, and we briefly discuss their viability for the objectives we have set for ourselves.

Viewed in the abstract, the principal theme of the present paper is that of modelling a simulation scenario as causal means-ends reasoning process. This scenario concept was first proposed in a paper published in the Simulation journal in 2022.[14] Here, causal theories are used for assessing the relative merits of different courses of action a trainee might pursue in his efforts to contain a situation. The causal theory is used to reason *backwards* from a goal to the various ways of achieving it. Each course of action that accomplishes the goal is assigned a score that reflects the quality of the trainee's decision making and instrumental reasoning. In the present paper, in contrast, the process is turned around; we reason *forwards* from actions to the future state of affairs caused by them.

There are several non-logical formalisms on offer with a proven track record in simulation modelling. The Discrete Event System Specification (DEVS)[4] is a prominent representative for this group. It is a formalism for modelling and analyzing hybrid continuous state and discrete event systems. An algorithm for executing a DEVS specification was proposed in the Simulation journal in 1987.[15] Since then, many extensions and variants have been explored, for instance P-DEVS[16] for simulation of parallel processes and RT-DEVS[17] for realtime simulation. A DEVS specification is expressed in a set-theoretical language that fits the general structure of deterministic, causal systems in classical systems theory.[18] This set-theoretical foundation has many advantages. To mention but one, mathematical functions of any kind, for instance differential equations, can without further ado be incorporated into the framework to describe the evolution of simulation state.

One property of our logic-based approach that contrasts favourably with DEVS, is that it furnishes the specification language with a well-studied[8,19–21] formal model theory that induces a concept of *logical consequence*: A statement follows from a specification if it is true in all models of that specification. In logic-based systems generally, logical consequence forms the basis for *query answering* understood as precise request for information made to a database or information system. Indeed, the central topic of the present paper of predicting futures from a causal theory *is* an example of query answering. Another example would be to retrieve the previously mentioned courses of action that accomplish a given goal.[14] It should be emphasized, though, that model theory is not necessarily a hard line between DEVS and logic-based approaches. It has been shown that DEVS can be given a formal semantics in terms of temporal logic.[22] That line of research does not appear to be a very active or developed area at the time of writing, however.

Conversely, DEVS has many interesting features that a logic-based approach based on answer set programming would struggle with. For instance, DEVS, through its internal transition functions, can apply any mathematical function, including real-valued functions of real variables, to the transformation of a state. Currently, most implementations of answer set programming only offer integers, though external functions defined in Python or Lua can be called to compute a value. An exception is the language s(CASP),[23] which is essentially Prolog with answer set semantics. As a dialect of Prolog, s(CASP) inherits floating point numbers from it. Programming in s(CASP) is rather different from answer set programming, however, and a closer comparison must be left for another occasion.

Beyond DEVS there are semi-formal specification frameworks such as the high-level architecture (HLA), which is an object model-based simulation protocol standard[24] that can be implemented in various simulation frameworks and platforms. HLA is associated with an abstract specification format called the Base Object Model (BOM) format,[25] and more expressive abstractions have been proposed in the form of ontology-based semantic extensions to the BOM format.[26] Such ontologies are currently under development for both military and civilian domains,[27–31] where they are also studied as prerequisites for the MSaaS vision.[32,33] To enable exercise managers, who may be non-technical experts, to compose simulations from simulation services, the services must have semantic and abstract non-technical, but machine-readable, service descriptions.[34,35] All this is for the rapid composition of object simulations and their interactions across various simulation platforms.

The field of formal specification and verification, also deserves to be mentioned under the rubric of related work, and the question arises as to whether any of its methods apply to our discussion. Traditionally, these

methods apply, in some way or another, to deriving correct programs from formal specifications via provable stepwise refinement. This is principally different from the present focus which is to specify scenarios and employ causal reasoning about them (and not the programs implementing them). However, there are also underlying similarities that are worth investigating at a later stage, and the satisfiability modulo theories (SMT) formalism, which has greater expressiveness than ASP, has been used in conjunction with stepwise refinement for B specifications.[36–38]

Since our scenario specification concept is based on declarative logic, it is independent of any particular simulation platform. Platform and implementation independence is a major concern in the modelling and simulation community, and is essential when combining several distributed simulations in a shared so-called Life, Virtual, Constructive (LVC) environment.[39,40] Various players on various simulation systems can play in person (with instrumentation) or move entities around either as first person players or as leaders of semi-automated entity aggregates. The HLA and BOM format mentioned above are central to this (with the already-mentioned limitations).

On the simulation platform side, software frameworks such as VR-Forces, Virtual Battle Space (VBS), Calytrix, MASA Sword and others, support the declaration and development of highly realistic domain-specific object behaviour and -interactions.[¶] Although there is support in these frameworks for scripting sequences of events, and even for expressing such scripts in terms of high-level and low-level operational commands,[44] there is no inherent notion of causality and means-end relationships. Hence there is no principled way to reason about the set of possible evolutions of a scenario as a function of the possible outcomes of actions.

Finally, the idea of factoring out the logic of a game to an answer set program is not in itself a novel one. The ThinkEngine[45][‖] is a framework for integrating an ASP solver into Unity games, much like our scenario event controller. The main difference is that ThinkEngine is a generic framework for delegating reasoning tasks to ASP, whereas our event controller is specific for the domain of causal reasoning. Our ScenarioEngine was designed specifically for supporting an exchange of actions for schedules of future events that inform the evolution of the simulation over time, see Section 6. As a domain specific piece of software the ScenarioEngine can assume much more about the relationships between the ASP code and the C# Unity environment. For instance, it is able to autogenerate an API for any given scenario description expressed as a causal theory, see Section 10. Moreover, ThinkEngine uses the generic EmbASP[46] framework as its interface to the answer set solver. We chose instead to use the Clingo C API, producing code that is both smaller, faster and (in our experience) easier to integrate with Unity in an OS-independent way. For this, we drew some inspiration from an existing library called clingo-cs.[47]

3 Answer set programming

Answer set programming (ASP)[13,48,49] is a language for knowledge representation and reasoning based on the answer set semantics of logic programs. As a member of the logic programming family of languages, an answer set program describes a problem with if-then rules of first-order logic, suitably restricted to ensure desirable computational properties. Other major logic programming languages include Prolog and Datalog.

Prolog and its dialect are languages based on *unification*, which is the algorithmic process of solving equations between symbolic expressions by searching for values to substitute for the variables in a rule. The computational process in ASP is quite different and involves two separate steps: first the program is *grounded*, which means that variables are systematically replaced in *all* rules in all possible ways to yield a variable free propositional program. Next, the propositional program is *solved* by generating all models that *satisfy* the rules. Hence, whereas languages based on unification searches for variable substitutions constrained by the set of rules of the program, ASP programs are solved by computing models represented as complete sets of true facts. Solving a computational problem in ASP thus means computing the models that satisfy the description of the problem as expressed by the rules of the program.

[¶] Viable technologies for emulating such behaviour include agent-based models[41], multi-agent systems[42] and probabilistic action networks[43]
[‖] https://github.com/DeMaCS-UNICAL/ThinkEngine

Anticipating subsequent sections of this paper, a logic program in the present paper will always be a description of a training scenario (a causal theory). The satisfying models will be sequences, or *schedules* as we shall call them, of events that follow logically from that scenario and a set of actions.

A lauded feature of ASP is that it achieves full *declarativeness*, meaning that the programmer does not need to give instructions on *how* to generate satisfying models. The actual computation of models is instead delegated to a generic *answer set solver*, which is usually a suitably modified satisfiability solver for classical first-order logic. This separation of the description, or the *what*, of a computational problem from the *how* of solving it is epitomized by Kowalski's famous motto Algorithm = Logic + Control.[50] Different logic programming languages have lived up to it to varying degrees.

The semantics of an answer set program is defined in terms of a carefully selected set of models that provides a declarative semantics for logic programs with *negation as failure*. This is the subset of the classical first-order models of the program known as its *answer sets* or *stable models*. Negation as failure is the inference pattern Kowalski describes as "the derivation of negative conclusions from the lack of positive information".[51] It is crucial for the present paper insofar as it can be used to express *default reasoning*. As explained in the introduction, default reasoning is the process of drawing conclusions on the basis of what one would reasonably assume to be true if nothing indicates otherwise. In colloqial language, it stands for our commonsense habit of jumping to plausible conclusions in the absence of evidence to the contrary, knowing that these conclusions may be rendered invalid by new information.

The present section recapitulates the essentials of the answer set semantics as first formulated in.[48] The interested reader should consult the list of references.[6,8,19,21]

The signature σ of an answer set program consists of *variables*, *object names* (also known as *constants* or *object constants*), *function symbols*, *predicate symbols* and *logical connectives*. The convention is that variable symbols are arbitrary strings of letters and numbers that start with an upper-case letter, while constants, predicate symbols and function symbols are strings that start with a lower-case letter. Object and function constants are used to construct *terms*, which are defined inductively in the usual manner:

Definition 1. *Term.*

1. *A variable X is a term.*
2. *A constant is a term.*
3. *$f(t_1, \ldots, t_n)$ is a term whenever f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms.*

A term is *ground* if it contains no variable symbols. An *atom* is either the boolean constants or $\top$, or a formula $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. The set of atoms over a signature σ will be denoted $\mathbb{A}$ – the signature being left implicit. If each term t_i is ground, then the atom is ground. A *literal*, denoted $l, l_i, \ldots$ is an atom or a literal prepended either by *strong negation* $\neg$ or by weak negation '*not*' aka. *negation as failure*. A set of literals is *consistent* if it does not contain a *complementary pair* $l, \neg l$ of literals.

When Γ is an ASP program, $lit(\Gamma)$ denotes the set of literals contained in Γ. The language of *extended logic programs*, consists of rules of form:

$$l \leftarrow l_1, \ldots, l_m, not\, l_{m+1}, \ldots, not\, l_n. \qquad (1)$$

If $n = 0$ then (1) is called a *fact*. The expression to the left of $\leftarrow$ is the *head* of a rule and the expression to the right of $\leftarrow$ its *body*. A rule is ground if each literal that occurs in it is ground. A program $gr(\Gamma)$ consisting of ground instances of all rules in Γ is called the *ground instantiation of* Γ. If r is a rule then $h(r)$ denotes its head and $b(r)$ its body. The comma-separated lists in the body of the schematic rule (1) denote conjunctions of literals.

Rules with non-empty bodies may be thought of as production rules: if the body of the rule is deducible from a program then the head of the rule may be added to the set of facts.

The difference between strong negation and negation as failure is important: an explicitly negated atom $\neg p(t_1, \ldots, t_n)$ is deducible from a program if that literal occurs in the head of some rule. In other words $p(t_1, \ldots, t_n)$ is asserted to be false under the conditions stipulated in the body of the rule. In contrast $not\, p(t_1, \ldots, t_n)$ is true in *absence of evidence to the*

contrary; that is, it is true if $p(t_1, \ldots, t_n)$ can *not* be deduced from the rules of the program. This difference is often illustrated with the following example attributed to John McCarthy:[49] consider the rule $cross \leftarrow not\ train$ expressing the maxim that "you can cross if you have no evidence that a train is coming". The rule $cross \leftarrow \neg train$, in contrast, captures the stronger requirement that "you can cross if you have positive evidence that no train is coming".

The semantics of answer set programs is defined for ground programs, and its full statement requires a fair bit of technical nomenclature. Suffice it here to outline the essentials: we start with the classical notion of an *interpretation* or *model*: a *model* I of a program Γ is a ground subset of $lit(\Gamma)$ that is closed under the rules of Γ, meaning that I satisfies the head of a rule of form (1) whenever it satisfies its body. The satisfaction relation $I \models \phi$ between a model I and a formula ϕ (rule or conjunction of complex literals) is defined as:

$$\begin{aligned} I &\models p \text{ iff } p \in I, \text{ for an atom } p \\ I &\models not\, p \text{ iff } p \notin I \\ I &\models \neg\, p \text{ iff } \neg p \in I \\ I &\models \phi, \psi \text{ iff } I \models \phi \text{ and } I \models \psi \\ I &\models \phi \leftarrow \psi \text{ iff } I \not\models \psi \text{ or } I \models \phi \end{aligned}$$

The reader should note the difference between the second and third item on this list.

The answer sets of a program Γ, as mentioned already, is a particular subset of the set of models defined above – intuitively it is the set of models that can plausibly be treated as representations of the beliefs or assumptions of a rational agent. This view is all but implicit in the following property of answer sets: An answer set for Γ is both closed under the rules of Γ and it is such that every true proposition according to it is derivable by applications of those rules. In other words, everything that follows from the rules is believed, and if it doesn't follow from the rules it is not believed.

Answer sets are easily identified for certain classes of simple programs. For instance, if a program has a finite set of rules and negation as failure does not occur, then it has a unique minimal model which is also its answer set–it is the single model that satisfies the set of rules with no gratuitous information. A program with negation as failure, on the other hand, may have more than one minimal model. For example,

$$\Gamma = \left\{ \begin{array}{l} b \leftarrow not\, a \\ a \leftarrow not\, b \end{array} \right.$$

has two minimal models $\{a\}$ and $\{b\}$.

Answer sets can be characterized as fixpoints equations of the form

$$M(\Gamma^I) = I$$

Here I is a model of Γ and $M(\Gamma^I)$ picks out the set of minimal models of a program Γ^I. The program Γ^I is the *reduct* of Γ relative to I. By definition, Γ^I contains a rule

$$l \leftarrow l_1, \ldots, l_m \tag{2}$$

for every rule of form (1) in Γ such that $\{l_{m+1}, \ldots, l_n\} \cap I = \emptyset$. In other words, the reduct Γ^I of I is a positive program obtained from Γ by eliminating all rules whose weakly negated premises are contradicted by I and removing all weakly negated premises from the remaining rules. The result is a program without negation as failure. Its unique minimal model is an answer set of Γ if it is consistent:

Definition 2. *A model I is an* answer set *of Γ if $M(\Gamma^I) = I$ and I is consistent.*

Answer sets as such are not in general unique as there may be more than one model that satisfies the equation of Definition 2.

Answer sets induce a consequence relation between extended logic programs and literals:

Definition 3. *A program* Γ entails *a literal l denoted $\Gamma \models l$ if l is satisfied by every answer set of* Γ.

Note that it follows from 3 that $\Gamma \models$ *not* l if l is not in any answer set of Γ. Two programs Γ_1 and Γ_2 are said to be equivalent if they have the same answer sets.

In the following we shall switch between the abstract ASP syntax defined in the present section, and a code-listing format that reflects the concrete syntax of Clingo. The former will be used in proofs and formal definitions, whereas the latter will be used to discuss examples of executable ASP code.

4 The concept of a causal theory

A causal theory, as the term is commonly used in symbolic AI, [6,8,19,52] is a set of rules that defines a state transition diagram modelling the effects of actions on the world (simulated or real). A causal theory views the world as a dynamic system or automaton whose paths correspond to courses of action and their ensuing successive transformations of an initial state of affairs. The domain specific language in which these diagrams are defined is commonly called an *action description language*. It allows a concise and mathematically accurate description of a particular system's states and of its state-action-state transitions. [6]

Example 1. *A pan without a lid on it is overheating on a stove in an industrial kitchen. The pan contains vegetable oil which will ignite and burn if left to itself. An astute employee who recognizes the danger may attempt to avert it, but even in a simple scenario such as this, his success depends on competence and training. Ideally, he would notice early that the pan is overheating and simply turn down the stove. But if he is new to the job, he may not recognize danger until the pan smokes profusely. There is still time to prevent a fire if, say, the employee acts quickly and both turns off the stove and puts a lid on the pan (we are supposing that the cooking oil has been brought so close to the point of ignition that the residual heat in the stove will set it off unless the pan is covered). To complicate matters, some actions are prudent if performed at the right time but dangerous otherwise. Take the action of turning on the fan to dissipate 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 the fan is on when the oil catches fire, the fire spreads to the fan.*[**]

Example 1 is illustrated in the state-transition diagram in Figure 1. Here, the boxes are states, that is, complete and consistent sets of facts (relative to answer set semantics), with a hyphen standing for the strong negation of a fact. Each arrow/transition is labelled with a pair consisting of an action and its duration. Either but not both of these elements can be omitted. When the action is omitted, the arrow says that the source state evolves independent of agency into the target state over the course of the duration interval. When the duration is omitted, the arrow expresses that the target state is an instantaneous effect of performing the given action in the source state.

5 The action description language $\mathcal{AL}$

The present paper adopts the general representational strategy of the language $\mathcal{AL}$ and its dialects. [5–7]

It will serve as a point of departure for the more general $\mathcal{ALID}$ language in Section 7, which is specifically tailored for the deductive use of causal theories for scheduling events. Some features of the full $\mathcal{AL}$ language[††] have been omitted for simplicity, since we will not use these features.

Formally $\mathcal{AL}$ is parameterized by a sorted signature containing

- a set of function symbols partitioned into fluent symbols, action symbols and the arithmetic addition operator $+$
- the set of relation symbols (with associated arities) $< /2, step/1, holds/2$, $action/1, fluent/1$ and *occurs*$/2$.
- a set of constant symbols sorted into object constants and integers.

An *action* is a ground term formed from action symbols and object constants. Similarly, a *fluent* is a ground term formed from fluent symbols and object constants. The former will be denoted $a, a_1, a_2, \ldots$ and the latter with $f, f_1, f_2, \ldots$.

Conceptually, an $\mathcal{AL}$ action description/causal theory is a calculus axiomatizing the effect of actions in terms of properties that can be true of an object (or tuple of objects) at one point in time and false at another. To emphasize that we are talking about fluctuating property ascriptions whose truth is relative to a time, we will follow the established terminology [6,9,53] and refer to these properties as *fluents*. In $\mathcal{AL}$ this relationship between properties, time points and truth values is *reified*, meaning that it

[**] See [14] for a fuller discussion of this scenario in the context of simulation-based training.

[††] This applis to the distinction between inertial and defined fluents as well as to the closed world assumption for the latter

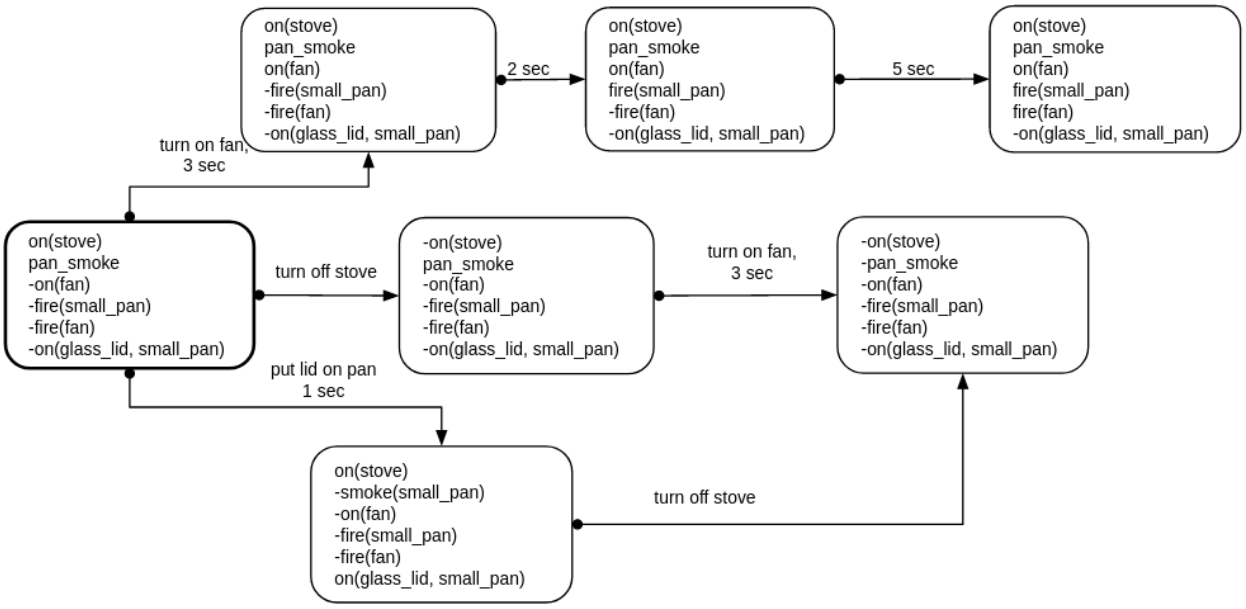

Figure 1. The kitchen example. The state drawn with a thick line is the initial state.

is stated explicitly, using the binary predicate $holds/2$. An atomic formula $holds(f, i)$ states that the property f, is *true* at *time i*, whereas $\neg holds(f, i)$ expresses that f, is *not true* at *time i*. When reasoning about causal relationships, the times at which a fluent changes from true to false are of primary interest. It is natural to think of these changes as *events*; the event of a property beginning to hold and the event of a property ceasing to hold. Any temporally ordered sequence of negative and positive *holds*-statements expresses a possibly empty set of events as so understood. In the following, when we speak of deducing a schedule of future events, it is such a linear sequence of positive and negative *holds*-statements that we have in mind.

As regards actions, they are intuitively atomic (i.e. non-complex) doings of an unspecified agent. In analogy to the treatment of fluents, actions are reified and recorded with the predicate $occurs/2$. A formula $occurs(a, i)$ states that the action denoted by a is performed at *time i*.

An ASP encoding of an $\mathcal{AL}$ causal theory (henceforth, when we talk about $\mathcal{AL}$ we shall mean the ASP encoding of it) has two parts: a *generic part* governing th general causal logic, and a *domain specific part* axiomatizing the specific causal relationships that are characteristic of the application domain. The generic part consists of

- a step declaration $step(0..n)$, where n is an integer.
- positive and negative inertia axioms:

$$\begin{aligned} holds(f, i+1) \leftarrow\ & holds(f, i), \\ & not\ \neg holds(f, i+1), \\ & fluent(f), \\ & i < n. \end{aligned} \tag{3}$$

$$\begin{aligned} \neg holds(f, i+1) \leftarrow\ & \neg holds(f, i), \\ & not\ holds(f, i+1), \\ & fluent(f), \\ & i < n. \end{aligned} \tag{4}$$

The step declaration sets the timeline and horizon for the causal theory, allowing it to evolve up to and including $step(n)$, but no further.

The inertia axioms give a formal expression of the common sense rule that what is true at one point in time can be assumed to be true in the future unless

evidence to the contrary can be inferred at the preceding timestep. The absence of evidence to the contrary is expressed with negation as failure as the unprovability of the opposite, where the opposite of an assertion is its *strong* negation. The combination of strong negation and negation as failure in the literal $not\ \neg holds(f, i+1)$ can thus be read as a possibility operator saying that fluent f may, *for all we know*, hold at step $i+1$. That is, $\neg holds(f, i+1)$ cannot be proved, so $holds(f, i+1)$ cannot be disproved, whence it is *possibly* true. To simplify the formal description, let $\pm h(f, i)$ be any element in $\{holds(f, i), \neg holds(f, i)\}$. The domain specific part of a causal theory encoded in $\mathcal{AL}$ consists of

- action declarations $action(a)$ for each action term a,
- fluent declarations $fluent(f)$ for each fluent term f,
- causal laws,

$$\begin{aligned} \pm h(f, i+1) \leftarrow \pm\, & h(f_1, i), ..., \pm h(f_n, i), \\ & occurs(a, i), \\ & i < n. \end{aligned} \tag{5}$$

- and, state constraints

$$\pm\, h(f, i) \leftarrow \pm h(f_1, i), ..., \pm h(f_n, i). \tag{6}$$

Causal laws capture the effects of actions on fluents. State constraints capture atemporal functional relationships between fluents that do not depend on actions. Since they are atemporal they do not increment time.

One causal law captures one effect of an action on a fluent. The action in question appears in the antecedent of that law, possibly together *a context of application* for that action consisting of other fluents. The effect of the action in this context is deduced by modus ponens by detaching the fluent statement in the head of the given causal law and adding it to the stock of inferred facts.

It is important to bear in mind the difference between a causal theory, on the one hand, and the actions it is applied to on the other. Although a causal theory axiomatizes the effect of actions, and thus refers to actions in the antecedents of domain specific rules, it is not a theory that says anything about which actions are *actually* performed. A causal theory is a set of general rules, action instances are specific facts. A single causal theory can predict the consequences of any number of actions on any number of occasions, provided it contains rules for them of course. Therefore, in order to actually deduce facts about the current or future state of the world from a causal theory, a history of actions must be provided as data for the causal laws to apply. In the following we shall refer to such histories of actions as *narratives*:

Definition 4. *A narrative is a sequence of actions* $occurs(a_i, m), \ldots, occurs(a_k, n)$ *where* $m, \ldots, n$ *is a linear ordering of integers.*

Note also, that due to the use of the'for all we know' locution in the inertia rules, domain specific rules apply only in the absence of evidence to the contrary. They are thus exception-allowing rules with a merely tentative validity in the sense already explained. To take an example, a pan with vegetable oil sitting on an overheated stove can be assumed to ignite (default rule) if, say, there is no evidence that it has a lid on it (exception). $\mathcal{AL}$ was designed to formalize precisely this kind of uncertain qualitative inference.

Peeking ahead to the next section, we shall exploit the defeasibility of $\mathcal{AL}$ rules, combined with the reification of fluents and actions, to define a simple protocol for passing information back and forth between a simulation environment and an answer set solver.The simulator will supply the narrative as data for the causal theory, and the causal reasoner will respond with a schedule containing future events that spell out the consequences of that narrative for the simulated scenario or world.

However, $\mathcal{AL}$ as it stands is not sufficiently expressive to support this protocol. In particular, it has three expressive limitations that must be removed. What they are will become clearer by describing the protocol from an abstract point of view.

6 A stateless protocol for simulation event control

As explained in the previous section, properties of objects and occurrences of actions at different points in time are stated explicitly using the two predicates $holds/2$ and $occurs/2$.

In both cases the second argument is an integer that denotes the timepoint at which a fluent holds or an action occurs. This technique of reification, as it is commonly

called, involves a semantic ascent, one might say, to a meta-language that talks *about* properties, actions and time, allowing general temporally extended relationships – notably inertia and causality – to be expressed.

One of the principal innovations of the present paper is to view this pair ($holds/2$, $occurs/2$) of predicates as an interface between, on the one hand, a system that emits actions and consumes state information, and, on the other hand, a system that consumes actions and emits state information. We have chosen Clingo and Unity for these respective components, but the concept is perfectly general; any answer set solver and any simulation platform may in principle be used.

The idea in a nutshell is to have the simulation system report a narrative in the sense of Definition 4 to the causal reasoner, upon which the causal reasoner responds with a schedule containing future events that spell out the consequences of those actions in the simulated scenario. For instance, if in a simulation of the kitchen example in Figure 1, the simulator sends a narrative to the reasoner according to which an agent first pours water onto a burning pan and then turns the fan on, then the causal reasoner will respond with a sequence of events, each being expressed by a *holds*-statements, in which there is a burning fan in the future. The repetition of this exchange of narratives for schedules constitutes a loop that acts as the driver for the evolution of the simulation scenario. Figure 2 provides an illustration.

This protocol is stateless. The causal reasoner is not expected to remember past actions, but receives a narrative containing all the actions that have been performed up to that point from the simulator. Each action bears a timestamp that indicates on the clock time of the simulator when that action was performed in the simulation environment. In general, the entire narrative is required for the computation of a correct schedule of events as each action may trigger different causal laws whose effects provide the context of application for other actions in turn. After all actions have been applied, the set of fluents that are detached constitutes a complete history of how the scenario evolves including a schedule of future events.

As explained in Section 5, the domain specific rules of a causal theory are default rules, which means that the computed schedule of events is a *default schedule*; future events will come to pass only if the agent does not act in a way to prevent it. Returning to the kitchen example, if the agent turns on the fan, then the fan will catch fire once the pan is burning. To put it in somewhat stilted language, there is a burning fan in the agent's default future. However, if he next acts with acumen by putting a lid on the pan in time to prevent the vegetable oil from igniting, then that future has been changed. In the following, we shall distinguish between a *schedule* which will be understood as a linearly ordered list of the start and end times of fluents, and the underlying *timeline* itself.

Anticipating the material in Section 10, the protocol in Figure 2 is implemented in the following schematized manner: the ScenarioEngine, which is implemented for Unity, reads the causal theory and generates a field for each fluent in a C# event handler object. The value of that field is a method that changes the state of an associated game object in the simulation, for instance by simulating flames in a pan. This method has to be coded by hand, obviously. The simulation begins when the ScenarioEngine calls Clingo with a possibly empty narrative to obtain a schedule of the future events of the simulation. These events are deduced from the causal theory and take the form of *holds*-statements. They are now associated through the C# handler object with actual methods on game objects. The timestamp in the *holds*-statement tells the ScenarioEngine when the corresponding method is to be called. Thus, the fluents in the deduced future in effect assert what should be true when *in the simulation*, for instance that the game object representing the pan ought to be burning two minutes into the simulation. The ScenarioEngine upholds its part of the protocol by calling that method at the appropriate time – unless, that is, that future is overridden by an intervening action, in which case a new narrative is sent and a new schedule received.

We said towards the end of Section 5 that $\mathcal{AL}$ has three expressive limitations that makes it unsuitable, as it stands, for this exchange. These limitations can now be seen in sharper relief.

The first is that the timeline, as given by the $step$-declaration of $\mathcal{AL}$, is postulated *a priori*. Hence, an $\mathcal{AL}$ theory itself states how long a scenario can go on for and when it has to end. Clearly, though, a simulation exercise may in general extend into an arbitrarily distant future, depending on the speed and acumen with which

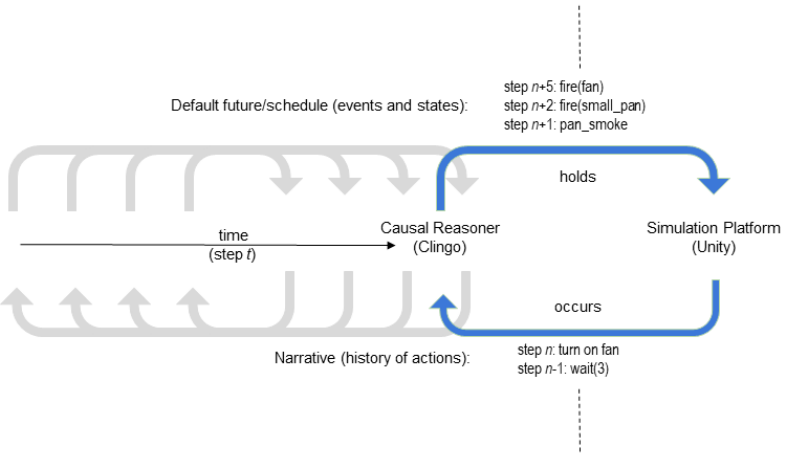

Figure 2. A stateless protocol for mediating between ASP scenario specifications and simulation platforms.

the trainee acts. He may choose to loiter indefinitely come what may, he may make abortive attempts to resolve the situation or he may pursue futile strategies. It follows that a suitable causal theory must be agnostic with respect to how much time the scenario will take to evolve, it cannot simply postulate a predefined end-point as $\mathcal{AL}$ does.

A second and related limitation concerns synchronization. A causal theory in $\mathcal{AL}$ postulates the timeline and the final time point independently of what happens in the simulation. But, the actions of the player in the simulation environment may fall within this time span or they may not. If they don't, then the effects of those actions won't either. The point is fairly obvious; it is really what happens in the simulation that should determine the timeline of the simulation. Therefore, a causal reasoner should *receive* the timeline as an *influx* from that environment. More specifically, it needs *infer* the timeline of past and future events from the narrative it receives from the simulator, where the timestamp of the latest action in the sequence can plausibly be taken to denote the present. This entails that time cannot be stipulated ahead of time as in $\mathcal{AL}$, but must be described by axioms that relate a narrative to past and future events quite generally. Only in this way can the deduced default future relate to and schedule events in terms of the clock time of the simulation.

The third and final limitation of $\mathcal{AL}$ from the perspective of the proposed protocol is that actions do not have *durations*. It is, for instance, not sufficiently expressive to capture the diagram in Figure 1. In $\mathcal{AL}$ all time increments are uniform; all causal laws increment time by a single unit, whereas state constraints do not increment time at all. Clearly, however, the duration of actions is not a concept that can be abstracted away from a simulation designed to exercise good judgement and decision making. Will the fire spread to the cabinets or the vent shaft first? How long will it take to put out each of them? Is there time to retrieve both the dry chemical extinguisher and the powder extinguisher from the panel on the back wall? If not, which one should the trainee prioritize, and so on. These choices depend on the duration of actions and events, which conceptually ought to be part of the causal theory and the default schedules deduced from it.

Taking stock, these limitations are be compressed into the following two desiderata for a more expressive version of $\mathcal{AL}$: Firstly, such a language should have the expressive resources to represent durative actions and events, and in particular to differentiate continuant and momentary events.[‡‡] Secondly, the language should be axiomatized to

[‡‡]Although this particular division is not made in the $\mathcal{AL}$ language (which instead has the type *defined fluent* for fluents that are defined

infer time from any narrative that is passed to it. That is, a causal theory formulated in this language would derive the schedule of future events from a given narrative and the durations of events given by ts causal laws.

7 The more general $\mathcal{ALID}$

For ease of reference, our proposed generalization of $\mathcal{AL}$ will be called $\mathcal{ALID}$ for "$\mathcal{AL}$ with *i*nferred time and *d*urative actions and events". The signature of $\mathcal{ALID}$ is a strict superset of that of $\mathcal{AL}$ containing the following additions:

- the fluent symbols are partitioned into three sorts; inertial and momentary. The former contains a possibly empty subset called the kinetic fluents,
- the $fluent/1$ predicate is replaced with $fluent/2$ to allow fluent declarations of different sorts. Two new relations $step\_between/2$ and $physical\_object/1$ are added to the set of relation symbols, and
- constants $inertial, momentary$, and $kinetic$, are added to the set of constants symbols to act as names for the corresponding fluent sorts.

The subdivsion of the set of fluents into different sorts reflects the necessity in any realistic model of a simulation scenario for distinguishing between, on the one hand, properties that are subject to the law of inertia, and thus by default propagated forward in time (e.g. the fan being on), and properties that do not endure in this way (e.g. a fire splash).

The subset of inertial fluents called kinetic fluents is a novelty of the present paper, justified in part for technical reasons. It serves the purpose of marking those inertial fluents the simulation environment does not need to register a handler for (where a handler is understood as a user defined method on a game object. See Section 10). Conceptually, a kinetic fluent represents an object property whose changes are fully accounted for by the physical simulation of the simulation environment itself – no need for ASP reasoning. A typical example (hence the terminology) would be the location of a box after it has been moved, or, say, the door angle after a door has been opened. The simulation environment does not need to consult the reasoner to infer the new location of the box or the new angle of the door. Kinetic fluents are inertial fluents, that is, they are subject to inertia and are thereby by default propagated into future state of the simulation by the reasoner. Yet, since they do not need to be processed by the simulation environment, they will be filtered out of the schedule and thus are not passed to the simulation environment. See Section 9 for illustrative examples.

We defer the discussion of the new $step\_between/2$ predicate to the next subsection, as it is best explained in the context of the generalized inertia axioms of $\mathcal{ALID}$. For similar reasons, we postpone a detailed discussion of the predicate $physical\_object/1$ to Section 10

7.1 The generic part of an $\mathcal{ALID}$ theory

$\mathcal{ALID}$ is technically an order-sorted logic,[54] that is, it is a fragment of first-order logic with sorted terms where the sorts are ordered into a hierarchy. In our case this hierarchy is expressed by one rule

$$fluent(inertial, f) \leftarrow fluent(kinetic, f). \quad (7)$$

subsuming kinetic fluents under the inertial ones.

To allow the timeline of the default future to be inferred from the narrative passed to the reasoner, $\mathcal{ALID}$ axiomatizes time replacing the simple step declaration $step(0..n)$ of $\mathcal{AL}$, with *step rules*.

- step rules:

$$step(0). \quad (8)$$

$$step(i) \leftarrow holds(f, i). \quad (9)$$

$$step(i) \leftarrow \neg holds(f, i). \quad (10)$$

$$step(i) \leftarrow occurs(a, i). \quad (11)$$

Rule (11) extracts timestamps from the narrative that is passed to the reasoner in the form of a sequence of $occurs$-atoms. More specifically, it makes *steps* out of the actions indices, that is, it converts action indices into timepoints represented in the way the reasoner expects. The effect is to synchronize the schedule deduced by the reasoner with the clock of the simulation, where the

in terms of other fluents), it is a straightforward modification logically speaking, and a distinction frequently found in the literature.

most recent action represents the current moment. The rules (9) and (10) ensure that the cumulative effects of the narrative is propagated into the future by having $(\neg)holds$-statements induce new timesteps, incrementing the time index with an arbitrarily long interval reflecting the duration of the last action and its effects. Finally, (8) is a ground atom and thus only a rule in a degenerate sense of course. It is an example of what we shall call a *milestone*, which is a ground atom of form $step(i)$ where i is an integer. A milestone refers to a fixed point in the timeline of a scenario, most obviously, but not exclusively, its starting point 0. Milestones are used in connection with state constraints, and more generally with *static causal laws* (definition pending) to schedule events that are not caused by actions. An example would be a pan sitting on an overheated stove due to ignite at a predetermined fixed point in the future unless someone acts to prevent it.

The inertia rules in $\mathcal{ALID}$ are generalizations of the corresponding $\mathcal{AL}$ rules where the single unit increments in the heads of the latter are replaced with a suitably abstract notion of the *next* step on the timeline.

- generalized positive inertia axiom:

$$\begin{aligned} holds(f,j) \leftarrow\ & holds(f,i), \\ & not\ \neg holds(f,j), \\ & fluent(inertial,f), \\ & not\ step\_between(i,j), \\ & step(j), \\ & i<j. \end{aligned} \tag{12}$$

- generalized negative inertia axiom:

$$\begin{aligned} \neg holds(f,j) \leftarrow\ & \neg holds(f,i), \\ & not\ holds(f,j), \\ & fluent(inertial,f), \\ & not\ step\_between(i,j), \\ & step(j), \\ & i<j. \end{aligned} \tag{13}$$

As can be seen, the $\mathcal{ALID}$ rules (12-13) are qualified to fluents belonging to the sort *inertial* (this is also the case in *full* $\mathcal{AL}$), thus exempting momentary fluents from being propagated forward to the next timestep.

A suitably abstract notion of a next timestep relative to i in rules (12-13) is the unique later step j with no step between it and i. This definition of 'next' is encoded as the *absence* of a step between them using the following straightforward *remote successor axiom*:

- remote successor axiom:

$$\begin{aligned} step\_between(i,k) \leftarrow\ & step(i), \\ & step(j), \\ & step(k), \\ & i<j, \\ & j<k. \end{aligned} \tag{14}$$

The conjoined effect of rules (12) and (14), to repeat the point, is thus remove the +1 increments in the heads and bodies of rules (3) and (4) and replace it with a more abstract notion of temporal immediacy. This makes the inertia rules in $\mathcal{ALID}$ agnostic with respect to the exact duration of the time intervals across which facts are to be propagated. It thus allows any amount of time to pass between two successive states in the evolution of the scenario, depending on the variable duration of actions.

We add one more rule to the generic part of an $\mathcal{ALID}$ theory, namely the closed world assumption for inertial fluents:

- the closed world assumption:

$$\begin{aligned} \neg\, holds(f,0) \leftarrow\ & not\ holds(f,0), \\ & fluent(inertial,f). \end{aligned} \tag{15}$$

This rule, which is a singularly well-studied principle in logic-based symbolic AI,[6,8,55,56] converts the absence of evidence for the truth of a property at time 0 into a proof for its falsity at time 0. It axiomatizes the assumption that information about the initial situation is complete; initially it is known whether a fluent holds or not. If it holds, it is asserted, if it does not hold, its strong negation is asserted. The inertia rules propagates positive and strongly negative facts forward in time, making every subsequent state complete in this sense as well.

Recall that the inertia rules are themselves defeasible due to the presence of negation as failure in their bodies. Therefore, a fact, positive or negative, is only propagated forward in time if no contradicting evidence

can be inferred at the preceding step. The upshot of this interaction between the closed world assumption and the inertia rules is that the overall behaviour of strong negation in domain specific rules simulates negation as failure. This is not a strictly necessary feature of the representation, but it cleans up the language and simplifies the grammar as negation as failure is no longer needed in the domain specific part.

Default rules are an important and indispensable part of the protocol described in Section 6, as such rules are what makes it possible for an agent to interfere with the unfolding of events and change the future. Some concrete examples of this will be provided shortly.

7.2 The domain specific part of an $\mathcal{ALID}$ theory

The action declarations of an $\mathcal{ALID}$ domain specification are similar to those of $\mathcal{AL}$. Fluent declarations are generalized to sorts, using the 2-place version of the $fluent$ predicate:

- a fluent declaration is a formula $fluent(s, f)$ where s is a constant in $\{inertial, kinetic, momentary\}$ and f is a fluent of the corresponding sort.

Object constants in $\mathcal{ALID}$ must be explicitly declared as physical objects. That is,

- an $\mathcal{ALID}$ theory has a formula $physical\_object(c)$ for every object constant in its signature.

The reason for this has to do with the way the protocol from Section 6 is implemented. Briefly put, the C# ScenarioEngine-library needs to know the *ontology* of a theory so that it can generate handlers for every event that takes one or more objects from the domain as arguments. More about this in Section 10.

Turning now to the form of rules, instead of the $\mathcal{AL}$ distinction between causal laws and state constraints, $\mathcal{ALID}$ draws a distinction between *dynamic causal laws* and *static causal laws*:

- a dynamic causal law is a rule of form

$$\pm h(f, i+k) \leftarrow \pm h(f_1, i), ..., \pm h(f_n, i), \\ occurs(a, i) \tag{16}$$

  where k is a nonnegative integer.

- a static causal law is a rule of form

$$\pm h(f, i+k) \leftarrow \pm h(f_1, i), ..., \pm h(f_n, i) \tag{17}$$

  where k is a nonnegative integer and either i is declared as a milestone or f_j is a momentary fluent for some j.

The differences between $\mathcal{AL}$ causal laws and $\mathcal{ALID}$ *dynamic* causal laws are small but important. Firstly, the +1 increments in the causal laws of $\mathcal{AL}$ can be replaced with any integer in a dynamic causal law of $\mathcal{ALID}$, allowing for the expression of actions and events of arbitrary duration. Secondly, since the $\mathcal{ALID}$ timeline is an inferred left-closed interval from 0 with no predefined maximum n, there is no constraint on dynamic causal laws to stay within an upper bound. Hence, $i < n$ from rule (5) does not occur in rule (16).

As regards state constraints vs. static causal laws, the former are meant to capture atemporal logico-functional relationships, such as e.g. the fact that a box is sitting on top of a stack of other boxes entails that the former is *above* all of the latter.[6] In contrast, static causal laws, though they may be used to encode functional relationships as well, also cover causal relationships that do not express agency. These causal relationships, like actions, may have any duration.

The timestep i in the body of a static causal law is required to be a milestone whenever f is an inertial fluent (possibly kinetic). In other words, a static causal law that governs an inertial fluent requires a context in which there is a ground atom $step(i)$ declaring the integer i to be a point on the timeline that is to be deduced by the reasoner.

This has to do with grounding. Most ASP systems including Clingo compute answer sets by first generating a ground propositional program that does not contain any variables but has the same answer sets as the original program. That program is then passed to a modified satisfiability solver that computes the models that satisfy the program.[57]

When the time index in the body atoms of a static causal law is a milestone, that rule is already variable-free. This ensures that all static and dynamic causal laws, and thus the entire domain specific part of an $\mathcal{ALID}$ theory conforms to a condition called λ-restrictedness.[58] Intuitively, λ-restrictedness means that there is a mapping λ of the predicates of a program to the natural numbers,

called levels, such that all variables in a rule with a particular predicate in the head occur in the body of that rule in atoms formed from predicates of a strictly lower level. Every λ-restricted program is known to have a finite equivalent ground instantiation, even in the presence of functions with non-zero arity, such as the $+k$ functions in the heads of $\mathcal{ALID}$ causal rules.

Conversely, if we allow time variables in static causal laws, they will break the λ-restrictedness condition, which easily leads to infinite groundings and non-termination of the computation *when the fluent is an inertial one*. A straightforward example of this behaviour will be provided in the next section.

If the fluent is a momentary one, on the other hand, this is not an issue since a rule with a momentary fluent in its body cannot be reapplied to the steps created by the rule head, thus avoiding an infinite generation of steps. As we shall see, in this case the restriction to milestones can be lifted, allowing static causal rules with variables ranging over timepoints.

One residual question remains, which is the question of why the time index in the body of a static rule needs to be declared to be a step when it is a milestone. Why is it not sufficient to make sure that the rule contains no variables, when trivially such a rule has a finite grounding? The answer is that the step declaration itself is not needed for grounding but for solving: for a static causal law to be applicable at a timepoint i all the relevant inertial fluents in the body of that rule must be propagated from the past up to the point of application i. This is the responsibility of the inertia rules. Now, the inertia rules propagate fluents over steps only, not arbitrary indices. Hence if i is merely an integer then the fluents in the body of the static causal law will not be propagated to i and hence the rule will never apply. This fact, which is easy to overlook, will also be illustrated with examples in the next section.

8 On the form of $\mathcal{ALID}$ theories: Determinism

The language $\mathcal{ALID}$ is more powerful than it may appear, insofar as it can express chance events. Consider the following pair of rules:

$$holds(fire(fan), 0) \leftarrow \\ \neg holds(fire(pan), 0). \tag{18}$$

$$holds(fire(pan), 0) \leftarrow \\ \neg holds(fire(fan), 0). \tag{19}$$

The two rules, which are state constraints, form what is known as a *cycle through negation*. For the two-rule case, this means that the fluent in the head of the one rule, depends on the negation of the fluent in the head of the other. As mentioned in Section 3, answer sets are not in general unique. It is well known[6,8,19] that a cycle through negation may induce non-uniqueness in answer set semantics. In particular, rules (18 -19) can be satisfied by two different answer sets; one in which the *pan* is on fire when the simulation starts, and one in which the *fan* is on fire when the simulation starts.

This pattern can easily be repeated for general (meaning time-unspecific) causal laws in $\mathcal{ALID}$. Consider the following pair of rules:

$$holds(fire(cabinet), T+1) \leftarrow \\ occurs(empty(jug, pan), T), \\ not\ \neg holds(fire(cabinet), T+1). \tag{20}$$

$$\neg holds(fire(cabinet), T+1) \leftarrow \\ not\ holds(fire(cabinet), T+1), \\ occurs(empty(jug, pan), T). \tag{21}$$

They form a negative cycle representing that the cabinet may or may not catch fire if one pours a jug of water onto a (presumed) burning pan.

Having the expressive power to model chance events is arguably a good feature of $\mathcal{ALID}$ since it allows the scenario designer to add to the training-value of an exercise by introducing an element of unpredictability. However, chance events should only come from deliberate design decision. They should certainly not be side-effects of the causal logic itself. The reason for this is fairly evident: When a causal reasoner is responsible for driving the evolution of simulation scenario, it is important to know the precise conditions under which the current state will fork into alternative futures and yield several possible lines of development or plays. The cumulative effect of *unintended* indeterminacy across time will, by definition, be difficult to anticipate and will obfuscate and hamper the design process.

Yet, it is not immediately obvious that unintended indeterminacy is ruled out by the causal logic of $\mathcal{ALID}$, because to every causal theory belongs a generic part that contains a negative cycle. The negative cycle, as the reader can check for himself, consists of the pair of generalized negative and positive inertia axioms, (12-13).

Importantly though, the inertia axioms have one feature that distinguishes them from the cycles we have seen in this section: Their bodies contain a *complementary pair*. That is, the body of the one rule contains the negation of an atom in the body of the other. This is the pair $holds(f, i), \neg holds(f, i)$. Since they are complementary, both formulae can not be true simultaneously, so both inertia rules can never *apply* simultaneously. Contrast this with rules (20-21). There is no such pair in *their* bodies, so both rules can apply at the same time.

It is possible, though rather involved, to prove that complementary pairs makes negative cycles safe in this respect under certain natural conditions. As a particular instance of this phenomenon, it follows that the inertia rules will derive one unique future from a set of domain specific rules, provided the domain specific rules satisfy those conditions.

The first condition is, rather obviously, that the domain specific rules must not explicitly be used to model chance events, as in the examples of the rules (18-19) and the rules (20-21). If they do, the world will split into alternative futures. The inertia rules will apply to each alternative, but will not fuse them together. This is as it should be. As argued already, it gives the scenario designer the power to introduce an element of chance deliberately.

The second condition is that causal rules must not be retroactive – an action should not be allowed to have an effect in the past. $\mathcal{ALID}$ itself allows this, but for obvious reasons, such strangeness should never be a necessary ingredient of any realistic modelling task.

Finally, the causal theory must be consistent. This also, is not guaranteed by $\mathcal{ALID}$ itself, since $\mathcal{ALID}$ rules can have strong negation in heads, viz.:

$$\neg holds(fire(fan), 0) \leftarrow holds(fire(pan), 0). \tag{22}$$

$$holds(fire(pan), 0) \leftarrow holds(fire(fan), 0). \tag{23}$$

The problem with inconsistency is of course not one of too many futures, but too few; an inconsistent causal theory does not predict anything.

We shall call a causal theory that satisfies these three conditions *well-founded*. Well-founded causal theories are deterministic in the following precise sense:

Definition 5. *A causal theory Γ is deterministic if for every narrative A, $\Gamma \cup A$ has exactly one answer set given that Γ is consistent.*

The proof of this claim is provided in the appendix.

9 Example: Representing the kitchen scenario

In the present section we represent the kitchen scenario from Example 1 in ASP, Clingo syntax, to demonstrate how the different language features of $\mathcal{ALID}$ can be applied in practice.

The example is elaborated a bit to give it a slightly richer set of features: in addition to Example 1 we are assuming that there are wooden kitchen cabinets on each side of the fan. There is a jug of water, and there is a panel on the back wall with a dry chemical extinguisher designed to extinguish class A (ordinary combustibles such as wood), B (flammable liquids and oil), and C fires (electrical equipment and appliances). There are five things the trainee can do in this scenario, as represented by the action declarations in Listing 1

```
1 action(put(glass_lid, small_pan)).
2 action(turn_on(fan)).
3 action(empty(jug, small_pan)).
4 action(empty(jug, fan)).
5 action(turn_on(fan)).
6 action(apply(drychem), fan).
```

Listing 1. Actions declarations in the kitchen domain.

The respective effects of these actions are described in terms of fluents declared in Listing 2 – the typing will be explained as we go.

Compared with Example 1, an agent now has a slightly bigger repertoire of prudent and imprudent choices: he can put out a fan fire using the drychem extinguisher. Alternatively he can also do so by pouring water over

```
fluent(kinetic, on(
                    glass_lid,
                    small_pan)
                    ).

fluent(inertial, on(fan)).
fluent(inertial, pan_smoke).
fluent(inertial, fire(small_pan)).
fluent(inertial, fire(fan)).
fluent(inertial, fire(cabinet)).

fluent(momentary, fire_splash).
```

Listing 2. Fluent declarations in the kitchen domain.

the fan, but that water will spill onto the pan, so if the vegetable oil in the pan is burning a fire splash will ensue that causes the kitchen cabinets to catch fire as well. Of course, this is also the result if the agent pours water directly on the burning pan.

The entire set of possible evolutions of the scenario is axiomatized by the domain specific causal rules in Listing 3 ('SCL' and 'DCL' in the comments are short for static and dynamic causal laws respectively). There are two static causal laws conforming to the rule schema (17) and eight dynamic causal laws conforming to rule schema (16). Both kinds of rule can be sorted into two kinds; positive laws that cause a fluent to hold and negative ones that cause a fluent to cease. For instance, SCL 1 states the conditions under which the pan catches fire whereas DCL 1 expresses a sufficient condition for putting it out. Similarly DCL 3 gives a positive rule for when the fan will start burning whilst DCLs 4 and 5 gives two ways of extinguishing it; water and drychem. Momentary fluents, exemplified here by a fire splash happening, have only positive rules – DCLs 7 and 8 in this case. Since momentary fluents are not subject to inertia, they do not endure through time. Hence, it is not necessary to state the conditions for their termination.

Both static and dynamic causal laws can be default rules, meaning that they apply only unless certain recognized exceptional cases can be inferred to hold. For instance SCL 1 states that the pan will burn after five seconds if it is sitting on an overheated stove *unless* it has a lid on it. Similarly, DCL 4 states that the drychem extinguisher will put out a fire in the fan unless the pan is still burning. DCL 6 is a negative default rule according to which the fan will dissipate smoke unless the pan is still burning. All other rules are non-defeasible and express strict causal relationships for which there is no exception. This is typically but not necessarily true of kinetic fluents recording the movement of objects. Thus, according to DCL 2 putting a lid on the pan invariably causes the lid to be *on* the pan. This action *could* have been modelled as a default in which case the exceptions would correspond to *unsuccessful attempts* at putting on the lid. This is less plausible for DCLs 3, 7, and 8 that unlike DCL 2 does not model the action directly, but rather the causal consequences of it: given that the action in question is performed those consequences invariably ensue.

The ability to represent and reason about default rules is one of the selling points of Answer Set Programming in particular and logic programming in general. As explained in Section 7.1, $\mathcal{ALID}$ removes the need for negation as failure in domain specific rules in favour of strong negation in order to limit design choices and make it easier to express the logic of a scenario correctly. As the reader may recall, this is accomplished by combining the closed world assumption (15), converting the unprovability of a fluent into the provability of its strong negation, with the positive (12) and negative (13) inertia rules.

Whether strict or defeasible, all domain rules of $\mathcal{ALID}$ can express arbitrary durations. Unlike in $\mathcal{AL}$, the inertia rules of $\mathcal{ALID}$ do not commit to any particular uniform-length interval between events, but appeal only to an abstact notion of the next step on the timeline. This next step is always inferred, not stipulated beforehand, from the step rules (8-11) among which (9) and (10) create steps out of the indices in the heads of causal rules, projecting the effect into an arbitrarily distant future. This is true of dynamic causal laws, and also of state constraints and more generally static causal laws as well. For instance, SCL 2 states that the kitchen cabinet's catching fire is a causal *process or event* that gets going two seconds after a fire splash erupts from the pan. Similarly, DCL 6 states that the *action* of dissipating smoke using the fan takes 3 seconds. In Listing 3, all actions and events have been represented as taking *some* time. It might be tempting to think of simple actions such as putting the lid on the pan as instantaneous actions; the

```
%Time unit
#const second = 1000.

%Milestone.
step(5 * second).

%SCL 1: smoking pan ignites 5 seconds from start
holds(fire(small_pan), 5 * second):-
    holds(pan_smoke, 5 * second),
    -holds(on(glass_lid, small_pan), 5 * second).

%DCL 1: panfire is extinguished by a lid after 1 second.
-holds(fire(small_pan), T + 1 * second):-
    holds(fire(small_pan), T),
    occurs(put(glass_lid, small_pan), T).

%DCL 2: putting a lid on the pan takes 1 second.
holds(on(glass_lid, small_pan), T + 1 * second):-
    occurs(put(glass_lid, small_pan), T).

%DCL 3: turning on fan causes burning fan 5s later if pan is burning.
holds(fire(fan), T + 5 * second) :-
    occurs(turn_on(fan), T),
    holds(fire(small_pan), T).

%DCL 4: drychem puts out fan fire after 10s if pan is not burning
-holds(fire(fan), T + 10 * second):-
    -holds(fire(small_pan), T),
    occurs(apply(drychem, fan), T).

%DCL 5: water puts out fan fire after 2s if pan is not burning
-holds(fire(fan), T + 2 * second):-
    -holds(fire(small_pan), T),
    occurs(empty(jug, fan), T).

%DCL 6: fan dissipates smoke after 3s. if pan is not burning.
-holds(pan_smoke, T + 3 * second):-
    -holds(fire(small_pan), T),
    occurs(turn_on(fan), T).

%DCLs 7, 8: pouring water on fan or pan causes fire splash if pan is burning.
holds(fire_splash, T + 1 * second):-
    holds(fire(small_pan), T),
    occurs(empty(jug, small_pan), T).

holds(fire_splash, T + 2 * second):-
    holds(fire(small_pan), T),
    occurs(empty(jug, fan), T).

%SCL 2: a fire splash sets the cabinets on fire after 2s.
holds(fire(cabinet), T + 2 * second):-
    holds(fire_splash, T).
```

Listing 3. Domain rules in the kitchen scenario

lid is on in the instant that it is put on. This is possible, but it is a temptation better resisted since experience shows that the risk of inconsistency increases with the number of instantaneous actions. To see why, consider the following example: suppose the action of turning on the stove has been modelled as an instantaneous action; the stove is on the moment it is turned on. There is an exception, let's say, which which is when the fuse is blown. The cause of a blown fuse is when something draws too much power from the circuit. Suppose this too has been modelled as an instantaneous event; if all appliances are on at the same time, then the fuse is blown. It follows that if all appliances but the stove is on at a given point in time, then turning on the stove will cause it to be on and off at the same time.

The two rules SCL 1 and SCL 2 highlights the different treatment of respectively inertial and momentary fluents in static causal laws. Since according to Listing 2 the pan's burning is an inertial fluent, it follows by the provisos to rule (17) that the time index of SCL 1 has to be a ground step. It has to be declared as such, i.e. the integer in question must explicitly declared to be a step, since the inertia rules do not propagate fluents over integers generally. Compare SCL 1 with the flawed version in Listing 4: the latter does *not* license the inference to the burning of the pan, even if the lid is not on the pan. This is due to the fact that in this representation when the statement in Listing 4 line 2 and 3 are evaluated, T is just an integer, not a step. Therefore, the negative inertia rule will not propagate the absence of a lid from 0 to T for the simple reason that T is not yet a step.

As explained in the previous section, the milestone trivially makes SCL 1 λ-restricted, ensuring that it can only be instantiated in a finite number of ways (one way in fact, since it is already a ground rule).

SCL 2, in constrast, is not λ-restricted. Obviously there is no level mapping, i.e. no function, that assigns two different integers to the holds predicate, and therefore not true that every variable in the head of the rule is bound by a predicate from a strictly lower level in the body. Yet, since the time variable in the head of SCL 2 is the time index of a momentary fluent fire_splash in the body of that rule – in conformity to the proviso of rule (17) – SCL 2 still has a finite grounding.

9.1 Deducing schedules

In accordance with the protocol described in Section 6, it is the responsibility of the answer set solver to pass a schedule to the simulation environment to be used as a schedule for future events. Based on the domain specific causal rules of an $\mathcal{ALID}$ theory, that schedule is deduced by calculating the moments at which fluents begin and/or cease to hold. To that end, the predicate begins/2 in Listing 5 states that a fluent begins to hold at a particular point if it holds at that time and does not hold at the immediately preceding step. The predicate starts/2, is a thin veneer on top of begins/2 that filters out kinetic fluents.

Recall from the previous section, that kinetic fluents is a subtype of inertial fluent that the simulation environment does not need to register a handler for. An example from Listing 3 is the fluent on(glass_lid, small_pan): if an agent performs the action of putting a lid on the pan in the simulation environment at a particular time T, then, as per the protocol, the simulation environment sends occurs(put(glass_lid, small_pan), T) to the ASP reasoner as part of a narrative recording all actions from the start of the simulation. The simulation environment does not need to wait to be told by the ASP reasoner that this places the lid on the pan – and then place it there – since the physical simulation takes care of this on its own.

The duals to the predicates begins/2 and starts/2 is the pair of predicates ceases/2 and stops/2. Their definitions, which are obvious from the positive duals, have been omitted. They represent the *negative future* understood as the sequence of steps at which fluents cease to hold.

```
1  starts(F, T):-
2    begins(F, T),
3    not fluent(kinetic, F).
4
5  begins(F, T2):-
6    holds(F, T2),
7    not holds(F, T1),
8    not step_between(T1, T2),
9    step(T1),
10   T1 < T2.
```

Listing 5. The commencement of fluents.

```
1       holds(fire(small_pan), T):-
2           holds(pan_smoke, T),
3           -holds(on(glass_lid, small_pan), T),
4           T = 5 * second.
```

Listing 4. Flawed default rule

Now, in order to actually extract the schedule, and only the schedule, from the answer set of the $\mathcal{ALID}$ theory in question, we use Clingo #show-directives to query the answer set for the information we are after. Briefly put, a #show-directive instructs Clingo to return only the specified elements of the answer set and suppress all others. Hence, the simple query in Listing 6 returns the schedule that is implicit in the answer set.

9.2 Some sample plays

We have distinguished between *narratives*, which are sequences of actions linearly order by time, and *schedules* which are lists of assertions saying when fluents start to hold and cease to hold. In the protocol from Section 6, a narrative is passed from the simulation environment to the ASP reasoner, while the scheduled induced by it is returned to the simulation software. Henceforth, we shall refer to the union of a narrative and the schedule it induces as a *play*.

Even simple domain specifications such as that of Listing 3 may give rise to a large number of narratives and schedules. If the $\mathcal{ALID}$-theory in question is deterministic (cf. Section 8), then the number of narratives is an upper bound on the number of schedules. This upper bound is given by the following formula: where n is any finite number of inferred steps and a the number of action declarations in the domain specification, it is

$$\sum_{i=0}^{|a|}\left(\binom{a}{i}\times\binom{n}{i}\times i!\right)$$

From the point view of simulation-based training, very few of these plays will be interestingly different. Rather, it is natural to consider uniqueness only up to some equivalence relation, for instance up to sameness of schedule. That is, one may treat two plays as essentially similar (or perhaps equally good?) if the respective narratives induce the same schedule.

Of the 1545 plays induced by Listing 3 on a five step time-line, there are 205 different schedules. Many of those schedules in turn differ only with respect to the intervals between events, but not with respect to their ordering. In general, which plays one considers "the same" will depend on the uses to which one puts an $\mathcal{ALID}$ theory.

For the purposes of driving application state in a simulation environment, it does not matter much. As long as an $\mathcal{ALID}$ theory is deterministic in the sense of Definition 5, each narrative induces one and only one play by deduction from the domain specification and the generic causal logic. That deduction is typically computationally inexpensive, as it essentially amounts to forward chaining (most of which is accomplished already at the grounding stage by a technique known as *partial evaluation*[21]).

Listing 7 selects five of the plays induced by the domain specification in Listing 3 that exemplify features of $\mathcal{ALID}$-causal rules that we have discussed. All of them presuppose that the pan is producing smoke when the simulation begins, which is here assumed to be at time 0. In this play the pan starts burning after five seconds due to the milestone in line 2 of Listing 3 and the static causal rule SCL 1, which, being static, applies irrespective of agency. Since the agent refrains from doing anything, the pan simply continues to burn.

Since SCL 1 is a default rule, it is possible for an agent to interfere with the forecasted default future and prevent the panfire. In play 2, he does so in the best way possible: he puts a lid on the pan before it ignites, and he turns on the fan to dissipate the smoke afterwards. Due to the strict causal rule DCL 1, the panfire never breaks out, which in turn means that the default rule DCL 6 applies, the only exception to it being a burning pan. Hence, the pan stops smoking.

Time is of the essence. In play 3 the agent puts the lid on too late, and the panfire has time to erupt. The agent then makes the further mistake of starting the fan before

```
#show starts/2.
#show stops/2.
```

Listing 6. A query that generates a schedule.

Play 1: The agent does nothing.

```
starts(pan_smoke, 0).
starts(fire(small_pan), 5000).
```

Play 2: The agent puts lid on before pan catches fire, then turns on fan.

```
starts(pan_smoke, 0).
occurs(put(glass_lid, small_pan),4000).
occurs(turn_on(fan), 20000).
stops(pan_smoke, 23000).
```

Play 3: Too late with the lid, too quick with the fan.

```
starts(pan_smoke, 0)
starts(fire(small_pan), 5000).
occurs(turn_on(fan), 10000).
starts(fire(fan), 15000)
occurs(put(glass_lid, small_pan), 20000 second).
stops(fire(small_pan), 21000).
```

Play 4: The fan fire cannot be extinguished while pan is burning.

```
starts(pan_smoke, 0).
starts(fire(small_pan), 5000).
occurs(turn_on(fan), 10000).
starts(fire(fan), 15000).
occurs(empty(jug, fan), 30000 second).
starts(fire_splash, 32000).
starts(fire(cabinet), 34000).
occurs(apply(drychem, fan), 40000).
```

Play 5: Put out the panfire first, then apply drychem.

```
starts(pan_smoke, 0).
starts(fire(small_pan), 5000).
occurs(turn_on(fan), 10000).
starts(fire(fan), 15000).
occurs(put(glass_lid, small_pan), 30000).
stops(fire(small_pan), 31000).
occurs(apply(drychem, fan), 40000).
stops(fire(fan), 50000)).
```

Listing 7. Some plays with their narratives and timelines.

putting out the fire. Due to the strict causal rule DCL 3, this causes the fan to start burning. When eventually he does put a lid on the pan, the panfire stops, but the fan continues to burn.

In play 4, the agent again allows the panfire to erupt and then turns on the fan. The fan then starts burning. The fan grabs his attention and he doesn't think about putting out the panfire. Instead he concentrates on extinguishing the fire in the fan, deciding to pour water over it. Due to DCL 5, this would normally work except (line 30) in circumstances when the pan is still burning, as it is in this play. Therefore, water spills onto the pan creating a fire splash that causes the wooden cabinets on either side of the fan to catch fire. Still attempting to put out the fire in the fan, the agent decides to apply the drychem extinguisher to it, which again, would normally work due to the default rule DCL 4, except when the pan is burning. Thus the fan continues to burn, and so does the pan and the cabinets.

On a side note, the fire splash is never explicitly asserted to stop. This is a momentary fluent which as such is assumed to be transient. In particular, it can in general not be assumed to last until the next inferred step, since the next step can come in an arbitrarily distant future. Of course, one could add a step rule to make all momentary fluents last, say, a millisecond, if the ending times of momentary fluents were of interest.

Note also, still with regard to play 4, that DCL 7 and 8 implement two ways of causing a fire splash; the agent can pour water onto the pan or on the fan whilst the pan is burning. Swapping one action for the other in play 4 yields a play that is equivalent up to schedules in the aforementioned sense.

In play 5, the agent changes his routine a bit. Although he does not succeed in preventing the pan from catching fire, and although he makes the mistake of turning on the fan, he realizes at that point that extinguishing the panfire is priority number one. He puts a lid on the pan, gets the drychem extinguisher and applies it to the fan. Since the pan is no longer burning, the fire in the fan is quelched too (due to the default rule DCL 4), and the situation is contained.

9.3 Taking stock

Recent studies of game-based learning show that a key feature determining learning value is the story being played.[59–61] The examples above should serve to illustrate that $\mathcal{ALID}$ offers a powerful high-level abstraction for describing precisely this aspect of a game. Being carefully restricted, $\mathcal{ALID}$ rules are syntactically simple and easy to design. Since $\mathcal{ALID}$ is a fully declarative language, it abstracts away the control flow constructs required for software to perform an action or an event, allowing the scenario designer to focus exclusively on expressing the rules that govern the salient events. Although syntactically simple, $\mathcal{ALID}$ has the expressive power to describe complex scenarios. The combination of durative actions and default rules, in particular, is a powerful feature that makes it possible to describe a *multiplicity of possible stories* implicit in an initial scene very succinctly.

10 Integrating Clingo and Unity—an implementation of the stateless protocol

In this section, we describe an implementation of the stateless protocol of Section 6. In technical terms, this is handled by a piece of software we refer to as the *Scenario-Engine simulation event controller*. It was developed specifically for the Clingo answer set solver and the Unity Real-Time Development Platform, a prominent platform for developing computer games. Whereas the engine itself is written in C++, games are usually programmed in C# using Unity's scripting API. One can also import and reference .NET assemblies and native libraries, which is how we have implemented the ScenarioEngine.

10.1 Usage overview

We start with an overview of the steps one must take to integrate ScenarioEngine into a Unity game/project:

First, obtain the native Clingo library for the relevant platforms. This comes bundled with the Clingo distributions for Linux and MacOS, and a corresponding version for Windows can be built from the available source code quite easily.

Next, obtain the ScenarioEngine .NET assembly. It is easily built from the source code, which is available

through git at `https://code.nr.no/scm/git/ScenarioEngine`.

Then, create an empty .NET library project for the scenario with the ScenarioEngine assembly as a compiler platform source generator, and add an ASP file named ontology.lp to the scenario project containing the declarations of physical objects, fluents and actions of the $\mathcal{ALID}$ theory, see Section 7.2.

Build the scenario project; create a Unity project if it does not exist already; and import the native Clingo libraries, the ScenarioEngine assembly and the scenario assembly into the Unity project.

Create a text asset in the Unity project named causal_laws containing both the dynamic and the static causal laws of the $\mathcal{ALID}$ theory, see Section 7.2 (this text asset will contain ASP code, but adding the suffix .lp to its name will confuse Unity.).

Integrate ScenarioEngine into the Unity project by (i) calling the appropriate method in the scenario assembly when the player performs an action and (ii) registering handlers for every potential event. This will be explained in more detail below.

10.2 Example scenario

The code for the ScenarioEngine has been made public as part of this publication, see the git address above. We have also implemented a proof-of-concept Unity project that simulates the kitchen scenario in the running example. Due to its use of licensed Unity content, it cannot be published along with the ScenarioEngine, but here is a brief description: The Unity scene consists of a 3D kitchen with the by now familiar pan on a stove, a lid, and a fan, see Figure 3. A first person controller (FPC) allows the player to move around in the room and interact with these objects. Interactable objects are indicated by a hand icon that appears over an object when hit by the ray from the FPC.

Listing 8 shows the contents of ontology.lp for this example. The causal rules were shown in Listing 3.

10.3 Program structure

Figure 4 shows the native Clingo library and the two .NET assemblies imported for our example scenario. The ScenarioEngine assembly contains a static class Asp, which handles the communication between C#

Figure 3. The kitchen example in Unity

and the native library using the .NET P/Invoke API. The ScenarioEngine also contains code for generating scenario-specific assemblies at compile time. The KitchenScenario assembly is generated in this manner. Its source code only contains the scenario-specific ASP file ontology.lp (Listing 8), but the project file specifies that ScenarioEngine is to be used as a .NET compiler platform source generator. Thus, the classes and enum type in the KitchenScenario assembly are generated based on the contents of ontology.lp. Observe that in order to pull this off, ScenarioEngine not only calls Clingo at runtime, but at compile time as well. The code responsible for the compile time code generation is located in the namespace Gen. The version of ScenarioEngine imported into Unity can be built without this namespace.

The entry point to the simulation event controller from the Unity scripting API is the class ActionHandlerBehaviour, which inherits from a class MonoBehaviour of the Unity framework. It does three things:

- It loads the ASP text asset causal_laws from the surrounding Unity project.
- It provides access to an instance of the scenario-specific class ActionHandler.
- It handles the scheduling of future events using Unity's event loop.

10.4 Unity integration

The controller is integrated into the Unity game as follows: Whenever the player performs an action, a corresponding method on the ActionHandler object must be called. Conversely, handlers must be defined for

```
lid(glass_lid).
pan(small_pan).
physical_object(X) :- lid(X).
physical_object(X) :- pan(X).
physical_object(fan).
physical_object(cabinet).
physical_object(drychem).
physical_object(jug).

fluent(kinetic, on(Lid, Pan)):- lid(Lid), pan(Pan).
fluent(kinetic, on(fan)).
fluent(inertial, fire(Pan)) :- pan(Pan).
fluent(inertial, fire(fan; cabinet)).
fluent(inertial, pan_smoke).

action(put(glass_lid, small_pan)).
action(turn_on(fan)).
action(empty(jug, small_pan)).
action(empty(jug, fan)).
action(apply(drychem, fan)).
action(no_action).
```

Listing 8. Example ontology.lp

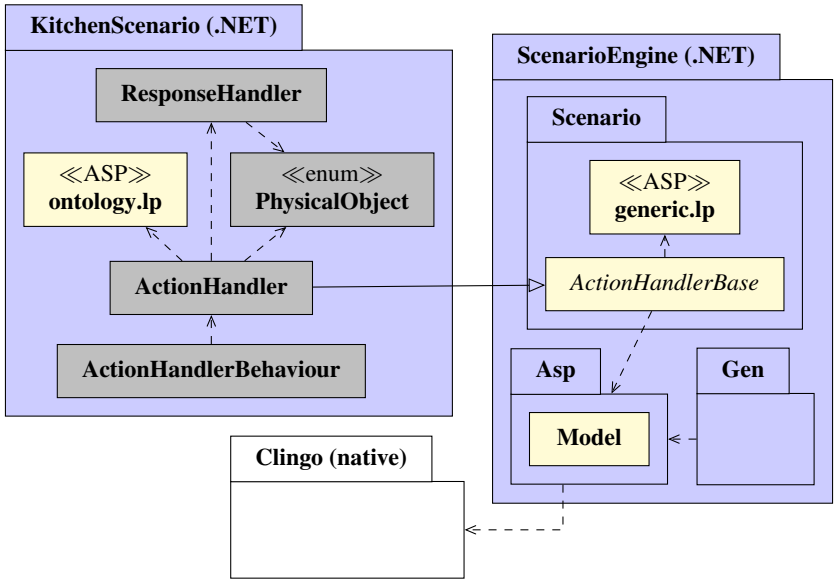

Figure 4. Architecture of a ScenarioEngine simulation event controller instance

every possible result event by setting the corresponding static properties of the ResponseHandler class. Listing 9 shows the signatures of these methods and properties in our example. Some of these take arguments from the enum type PhysicalObject, which contains these elements: glass_lid, small_pan, fan, cabinet, drychem, jug. ScenarioEngine also supports integer and string arguments.

When an action handler method is invoked, the ScenarioEngine reconsiders the events that are scheduled to happen in the future. First, every future event currently scheduled is cancelled. Next, Clingo is invoked in order to compute the new future of events that are caused by the narrative obtained by appending the new action to the history of previously performed actions. Unity sends everything Clingo needs to know in order to make this deduction. This is:

- the generic part of the causal theory, consisting of the causal logic in generic.lp and the ontology of objects, fluents and actions in ontology.lp,
- the domain specific causal laws and state constraint specified in the text asset causal_laws,
- the narrative consisting of the current and all previous actions encoded as occurs/2-facts.

The causal laws should be chosen so that this theory has one and only one model, cf. Section 8, in which case the action handler method ends by scheduling new future events based on the starts/2 and stops/2 facts in this model. Otherwise, an error is signalled.

10.5 Clingo interface

Potassco, who develops Clingo, does not provide a .NET API, but it is straightforward to invoke their C API via .NET P/Invoke. Since ScenarioEngine only needs a small part of this API, we implemented the communication from scratch rather than building on the existing systems discussed in Section 2.

Each stable model we get from the C API is essentially a list of abstract syntax trees representing literals. We preserve this structure on the .NET side using a form of Scott encoding so that we can process the models using pattern matching and a limited form of unification. We found this approach to be more flexible than the object-relational mappings used by EmbASP.

10.6 Taking stock

The core functionality of the ScenarioEngine is to generate a scenario specific .NET interface to the reasoning engine, which is easily integrated into the control flow of Unity 3D games and simulations. The interface is generated at compile time based on the ASP code alone. No "boiler-plate" code is needed for the integration; and if there are changes to the ASP scenario description that affect the interface, it will be discovered by the Unity programmers at compile time. This facilitates iterative scenario development by reducing the risk of introducing bugs.

On a higher level the function of the ScenarioEngine is to separate core game structures into those that have to do with the physical simulation and those that have to do with designing the story or stories that an initial scene can evolve into. More specifically, the ScenarioEngine factors the logic governing the evolution of a scenario out of the simulation development environment itself, making it an asset that can be managed, maintained and reasoned about independently. The idea is not unlike the decomposition expressed in Bob Kowalski's slogan *algorithm = logic + control*.[50] At the risk of overassimilation perhaps, one may think of a simulation scenario in terms of a similar equation *simulation = scenario + physical simulation*. By allowing the scenario designer to ignore all aspects having to do with the physical simulation, the ScenarioEngine does not require that person to know how to program in the simulation environment. Conversely, developers of the simulation environment are free to focus exclusively on the physical simulation and the behaviour of its objects, not needing to anticipate the twists and turns that a story may take. This can reasonably be expected to reduce the amount of programming it takes to configure a scene, and to increase the reuse value of that scene.

11 Concluding remarks and future work

In this paper we have developed a three-part concept for specifying and executing training scenarios on simulation- and game development platforms.

First, we have defined an action description language called $\mathcal{ALID}$, implemented in Answer Set Programming,

```
void Do__put(PhysicalObject x, PhysicalObject y)
void Do__turn_on(PhysicalObject x)
void Do__empty(PhysicalObject x, PhysicalObject y)
void Do__apply(PhysicalObject x, PhysicalObject y)
void Do__no_action()

static Action Starts__pan_smoke { get; set; }
static Action Stops__pan_smoke { get; set; }
static Action<PhysicalObject> Starts__fire { get; set; }
static Action<PhysicalObject> Stops__fire { get; set; }
```

Listing 9. Generated handler methods and properties

for commonsense default reasoning about actions and events in a simulation scenario. The language $\mathcal{ALID}$ is designed to support the computation of schedules of future events that are to be executed by some simulation platform.

Secondly, we have defined a stateless protocol to allow the computed schedule to be sent to the simulation platform for execution. The schedule contains a *default future of events*, understood as what will unfold given the actions taken so far in the simulation environment, and provided that nothing more is done. New actions can change the default future, propelling the evolution of the story along different lines.

Thirdly, we have implemented an efficient *simulation event controller* in the form of a C# library ScenarioEngine. It implements our stateless protocol using the ASP system Clingo[62] for causal reasoning. As a C# library it is suitable for use with the Unity 3D game development platform.

On the theoretical side, we have proved a theorem in the meta-theory of the proposed action description language that states that the schedule deduced from an $\mathcal{ALID}$ scenario description and an arbitrary narrative is unique, provided that the scenario description satisfies certain natural constraints; in a nutshell that 1) the domain specific rules do not express choice using cycles through negation, 2) that there are no retroactive domain specific rules that change the past, and 3) that the domain description is consistent. Determinism in this sense does not deprive the scenario designer of the means to model chance events and alternative future. It merely ensures that such events are not unintended side effects of the causal logic.

Future work divides into theoretical and practical. On the theoretical side it remains to prove that every $\mathcal{ALID}$ theory has a finite grounding. All our experimental work so far indicates that this is the case, but we have no proof to offer yet. The property is easy to prove for the domain specific part of an $\mathcal{ALID}$ theory, for reasons adduced above. It is not equally clear how to prove it for the generic part, since the generic part is not λ-restricted.

Still on the theoretical side, the computational complexity of checking whether an $\mathcal{ALID}$ theory is well-founded has yet to be determined. We conjecture that an algorithm can be found which is at least fast for the theories that will occur in practice.

Future work of a practical implementation-oriented nature includes optimizing the current implementation of the ScenarioEngine. In the current implementation every call to Clingo from Unity happens synchronously. This can cause the user interface to become unresponsive if the scenario is complex. Calling Clingo from a separate thread, which is what ThinkEngine does, is one possible solution. However, as this introduces a new set of challenges, it would perhaps be better to reduce the work Clingo has to perform. For example, we could remember the full state at the time of the previous user action rather than all the actions leading up to it, an approach that might perhaps be fruitfully combined with Clingo's support for incremental computation.

Finally, some stretch-goals naturally suggest themselves once one has a protocol and API for integrating ASP reasoning in a game development platform. What

ought to be a fairly low-hanging fruit is to use a causal theory to reason abductively from goals to the means of achieving them, instead of, as we do in this paper, deductively from actions to the resulting states. This could be used to extract plans for computer controlled agents to enable them to manipulate objects in a *purposeful and goal-oriented way* based on a means-end analysis of the scene.

Appendix: Proof of determinism

The notion of a well-founded causal theory was described informally in Section 8. Here we provide a mathematical definition and we prove that well-founded causal theories are deterministic in the sense given by Definition 5.

The proof exploits a well-know property of extended logic programs called *local stratification*,[19] which is a sufficient condition for the uniqueness of models.

11.1 Stratification

Let Γ be an extended logic program, let Γ' be its ground instantiation (as defined in Section 3) and define $>_+$ and $>_-$ to be the least binary relations on the set of literals in Γ' such that for every ground rule $l \leftarrow l_1, \ldots, l_m, \mathit{not}\, l_{m+1}, \ldots, \mathit{not}\, l_n$ in Γ':

- $l >_+ l_i$ for every $i \in \{1, \ldots, m\}$
- and $l >_- l_i$ for every $i \in \{m+1, \ldots, n\}$.

Now let

$$> \quad = \quad >_+^* \; ; >_- \; ; >_+^*$$

where * is is the reflexive and transitive closure operation and ';' is relation composition. We say that Γ is *locally stratified* if $>$ is a well-founded relation in the sense that there is no infinite decreasing chain

$$l_0 > l_1 > l_2 > \ldots .$$

If Γ is locally stratified, then $>$ must be acyclic. The converse holds if the set of literals in Γ' is finite. A *normal logic program* is an extended logic program which does not contain strong negation ($\neg$). The following result is well-known:[19]

Theorem 1. *If Γ is a locally stratified normal logic program, then Γ has exactly only answer set.*

This carries over to extended logic programs as follows. First, observe that strong negation can be emulated in logic programs with constraints by considering P and $\neg P$ to be separate predicate symbols and including constraints $\bot \leftarrow P, \neg P$ for every P. Next, observe that if Γ is a normal logic program and Γ' is a set of constraints, then the answer sets for $\Gamma \cup \Gamma'$ are by definition the answer sets for Γ that satisfy each contraint in Γ' (meaning that for every ground instance of the constraint, at least one of the premises should not hold in the answer set).

Since we do not allow (other) constraints in extended logic programs, we get:[49]

Theorem 2. *Let Γ be an extended logic program, and let $-\Gamma$ be the normal logic program where every strongly negated predicate symbol $\neg P$ in Γ has been replaced by a corresponding new predicate symbol $-P$. Then a subset $I \subseteq lit(\Gamma)$ is an answer set for Γ iff $-I$ is an answer set for $-\Gamma$ and $-I$ does not contain any contradictory pairs $\{P, -P\}$.*

We shall say that an answer set for $-\Gamma$ is *sound* if it does not contain any such contradictory pairs. Listing 10 shows that $-\Gamma$ may have both sound and unsound answer sets. Also observe that the way we have defined local

```
1  P :- not Q.
2  Q :- not P.
3  ¬Q :- not P.
```

Listing 10. Extended logic program Γ such that $-\Gamma$ has two answer sets, $\{P\}$ and $\{Q, -Q\}$

stratification, it applies to extended logic programs Γ in general – not only normal logic programs – and that Γ is locally stratified if and only if $-\Gamma$ is locally stratified.

11.2 Well-founded causal theories

In this section a *narrative* will be any logic program consisting entirely of ground facts of the form $occurs(f, t)$ where t is a nonnegative integer. Thus, $-A = A$ for every narrative A. The following is a generalization of $\mathcal{ALID}$ theories:

Definition 6. *We shall say that an extended logic program Γ is a* causal theory *if $\Gamma = \Gamma_b \cup \Gamma_i \cup \Gamma_c$ where:*

1. Γ_b *does not contain any (weak or strong) negation or any instances of the predicates 'step', 'step_between', 'holds' and 'occurs';*

2. Γ_i *consists of the generic part of any $\mathcal{ALID}$ theory, i.e. the rules pertaining to steps, inertia and the closed world assumption described in Section 7.1;*

3. Γ_c *consists of rules whose heads/conclusions all have the form 'step(t)', 'holds(f,t)' or '$\neg$holds(f,t)'.*

We shall refer to the rules in Γ_c as the *causal rules* of Γ.

It is easy to see that every $\mathcal{ALID}$ theory Γ is a causal theory in the above sense, with Γ_b containing the action and fluent declarations and Γ_c containing the causal laws, both dynamic and static.

For a causal theory Γ to be deterministic, the causal rules must be constrained. For this we need a new construct: Let Γ be a causal theory, let Γ' be its ground instantiation, let F be its set of ground fluents, and let $-F$ be a disjoint "mirror set" where each fluent is preceded by $-$. We define a labeled graph on $F \cup -F$ with edges $\overset{+}{\Leftarrow}$, $\overset{-}{\Leftarrow}$ and $\overset{!}{\Leftarrow}$:

First, we have $f \overset{!}{\Leftarrow} -f$ and $-f \overset{!}{\Leftarrow} f$ for all f. Next, let $l \leftarrow l_1, \ldots, l_m, \mathit{not}\, l_{m+1}, \ldots, \mathit{not}\, l_n$ be a rule in Γ' which is an instance of a rule in Γ_c and where $l \in \{\mathit{holds}(f,t), \neg\mathit{holds}(f,t)\}$ for some f. If $l_i \in \{\mathit{holds}(g,t), \neg\mathit{holds}(g,t)\}$ for some i and g (and the same t as in the head), then

- $f' \overset{+}{\Leftarrow} g'$ if $i \in \{1, \ldots, m\}$, and
- $f' \overset{-}{\Leftarrow} g'$ if $i \in \{m+1, \ldots, n\}$,

where

$$f' = \begin{cases} -f & \text{if } l \text{ is strongly negated,} \\ f & \text{otherwise.} \end{cases}$$

g' is defined similarly (in terms of g and l_i).

Definition 7. *A causal theory Γ will be called* well-founded *if its causal rules have the following properties:*

1. *There are no premises of the form 'not step(t)'.*
2. *Every timestamp in the body of a causal rule must be less than or equal to the timestamp in the head.*
3. *If the premises with timestamps in a causal rule are all weakly negated (with 'not'), then the timestamp in its head must be a nonnegative constant.*
4. *Whenever we have an infinite path*

$$x_0 \overset{y_0}{\Leftarrow} x_1 \overset{y_1}{\Leftarrow} x_2 \quad \ldots$$

in the graph defined above and it contains an infinite number of edges labelled '$-$' or '!', the path must contain both $f \overset{!}{\Leftarrow} -f$ and $-f \overset{!}{\Leftarrow} f$ for some f.

In this definition property 1 prevents conflicts with the basic rules for *step*, 2 expresses that no effect can happen before its causes, and 3 ensures that we do not have to consider negative timestamps. More importantly, property 4 puts restrictions on the negative dependencies in causal rules. This ensures that we avoid non-determinism even though we are using *not* to express default reasoning.

It is easy to see that every $\mathcal{ALID}$ theory has properties 1, 2 and 3. However, the following loop from the first example in Section 8 shows that 4 does not hold in general:

$$\mathit{fire}(\mathit{fan}) \overset{+}{\Leftarrow} -\mathit{fire}(\mathit{pan}) \overset{!}{\Leftarrow} \mathit{fire}(\mathit{pan}) \overset{+}{\Leftarrow} -\mathit{fire}(\mathit{fan}) \overset{!}{\Leftarrow} \mathit{fire}(\mathit{fan})$$

Remarks: As stated, property 2 of Definition 7 implicitly assumes that the ground instantiation of a causal theory will only have numeric constants at the positions of t and t' in atoms of the forms $\mathit{occurs}(f,t)$, $\mathit{holds}(f,t)$, $\mathit{step}(f,t)$ and $\mathit{step\_between}(t,t')$. For this to be true, the timestamp in the head of each causal rule must be an arithmetic expression of numeric constants and timestamp variables so that the grounding algorithm will not introduce symbolic expressions at these positions. The arithmetic functions available will vary between ASP solvers. A max-function is convenient to ensure that every causal rule satisfies property 2. Example:

```
holds(contract_valid, max(T1, T2)) :-
      occurs(buyer_signed, T1),
      occurs(seller_signed, T2).
```

Let Γ is a causal theory with a finite number of rules before grounding that only involve the arithmetic functions $+$, min and max. It is beyond the scope of this text but intuitively clear that it is decidable whether Γ is well-founded: Conditions 1 and 3 are trivial to check, condition 2 involves checking a finite number of cases, and condition 4 involves checking the finite set of loops

$$x_0 \overset{u_0}{\Leftarrow} x_1 \quad \ldots \quad \overset{u_n}{\Leftarrow} x_0 \ .$$

However, it is not clear whether checking conditions 2 and 4 is computationally efficient in general. Hopefully, an algorithm can be found which is fast for most causal theories that occur in practice.

11.3 The proof

We now set out to prove the following:

Theorem 3. *Every well-founded causal theory is deterministic.*

Let A be a narrative, let Γ be a well-founded causal theory. By Theorem 2 (and the role of grounding in the semantics of logic programs) we only have to prove that

$$\begin{gathered}\text{if } \Delta \text{ has a sound answer set,}\\ \text{then this is its only answer set,}\\ \text{where } \Delta \text{ is the ground instantiation of } -(\Gamma \cup A).\end{gathered}$$

This will follow easily from two lemmas concerning $\Delta\restriction_n$, which we define as the subset of Δ consisting of the (facts and) rules where all the timestamps are less than n.

Lemma 1. *Let n be a natural number. If I is an answer set for Δ or $\Delta\restriction_k$ for some $k \geq n$, then the restriction $I\restriction_n$ of I to timestamps less than n is an answer set for $\Delta\restriction_n$.*

Proof. Let I be an answer set for Δ. It is easy to verify that it is sufficient to show that for every rule $R \in \Delta \setminus \Delta\restriction_n$ the head of R does not occur in $\Delta\restriction_n$. If R is a causal rule, then this follows from condition 2 of the assumption that Γ is well-founded; and it is easy to see that the other rules in a causal theory also have this property.

The case when I is an answer set for $\Delta\restriction_k$ is similar if $k > n$ and trivial if $k = n$. □

Next, we will show that each $\Delta\restriction_n$ has the property we want for all of Δ:

Lemma 2. *Let n be a natural number. If $\Delta\restriction_n$ has a sound answer set, then this is its only answer set.*

Proof. The proof is by induction on n.

The grounding algorithm will ensure that all timestamps in Δ are nonnegative since we have assumed conditions 2 and 3 of Definition 7 and only allow nonnegative timestamps in narratives. (If in doubt, it is also easy to verify that removing all rules with negative timestamps from Δ yields a ground theory with the same answer sets.) Consequently, $\Delta\restriction_0$ does not contain any rules with timestamps. This leaves the (instances of) the facts and rules in Γ_b and the narrative A. Since these do not contain weak or strong negation, $\Delta\restriction_0$ has a unique answer set which is necessarily sound.

Now assume that the lemma holds for some n. We shall prove that it also holds for $n+1$, but first observe that

$$\begin{gathered}\text{every rule } R \in \Delta\restriction_{n+1} \setminus \Delta\restriction_n\\ \text{must have the timestamp } n \text{ in its head.}\end{gathered}$$

Since $R \notin \Delta\restriction_n$, it must contain at least one such timestamp. When R is a causal rule, condition 2 of Definition 7 means that this must also be the head timestamp; and (as mentioned in the proof of the previous lemma) it is easy to verify that the other rules of Γ also have this property.

Let I be a sound answer set for $\Delta\restriction_{n+1}$. Then $I\restriction_n$ is an answer set for $\Delta\restriction_n$ by Lemma 1; and it is sound since it is a subset of I. Thus, it is the unique answer set for $\Delta\restriction_n$ by the induction hypothesis. Next, we consider the cases $\mathit{step}(n) \in I$ and $\mathit{step}(n) \notin I$ separately.

First, assume that $\mathit{step}(n) \notin I$. Since Γ includes the rules (9) to (11), the timestamp n does not occur in I, and we have $I = I\restriction_n$. Let J be another answer set for $\Delta\restriction_{n+1}$. Then $J\restriction_n = I\restriction_n$ by Lemma 1 and the induction hypothesis. Suppose that $\mathit{step}(n) \in J$. Since J is the minimal model for the reduct $(\Delta\restriction_{n+1})^J$, it contains an atom a with timestamp n which is the head of a rule in $\Delta\restriction_{n+1} \setminus \Delta\restriction_n$ where each premise is true in J and no positive premise has timestamp n. Since the premises of this rule must also hold in I, we get $a \in I$, which contradicts the fact that I does not contain timestamp n. Thus, $\mathit{step}(n) \notin J$ and $J = J\restriction_n = I\restriction_n = I$.

Now, assume that $\mathit{step}(n) \in I$, and let E be the least ground extended logic program with the following properties:

(i) $I\restriction_n \subseteq E$ (included as facts),

(ii) *step*$(n) \in E$ (also included as a fact),

(iii) For every rule

$$l \leftarrow l_1, \ldots, l_p, \mathit{not}\, l_{p+1}, \ldots, \mathit{not}\, l_q$$

in $\Delta\restriction_{n+1} \setminus \Delta\restriction_n$ such that

$$l_i \in I\restriction_n \text{ iff } i \leq p$$
for each l_i that does not contain the timestamp n

E contains a rule which is identical except that we have dropped each premise (l_i or *not* l_i) that is either *step*(n) or does not contain n.

If we can show that E is locally stratified and that every answer set for $\Delta\restriction_{n+1}$ is an answer set for E, then I is the unique answer set of $\Delta\restriction_{n+1}$ by Theorem 1.

Let J be an answer set for $\Delta\restriction_{n+1}$. First, we see that J is a model for E. (i) $J\restriction_n = I\restriction_n$ (which is sound) by Lemma 1 and the induction hypothesis. (ii) If J is sound and *step*$(n) \notin J$, then J is the unique answer set for $\Delta\restriction_{n+1}$ by the argument above, constradicting the assumption that *step*$(n) \in I$. If J is not sound, then the contradictory pair in J must contain the timestamp n; and thanks to rules (9) to (11) we also have *step*$(n) \in J$. (iii) J satisfies each remaining rule in E since J satisfies the corresponding rule in $\Delta\restriction_{n+1}$ as well as every premise that was removed, cf. (i) and (ii). There is a similar relationship between the reducts of E and $\Delta\restriction_{n+1}$ with respect to J. Thus, J is an answer set for E by induction on the construction of the minimal model $M\big(\Delta\restriction_{n+1}{}^{J}\big)$.

Now assume that E is not locally stratified, which means that we have an infinite decreasing chain like this:

$$\begin{aligned} a_{0,0} &>_+ \cdots >_+ a_{0,n_0} >_- \\ a_{1,0} &>_+ \cdots >_+ a_{1,n_1} >_- \\ a_{2,0} &>_+ \cdots >_+ a_{2,n_2} >_- \cdots \end{aligned}$$

where we have emphasized each instance of $>_-$. We want to show that this contradicts property 4 of Definition 7, but first we must identify the predicate symbols in this chain.

Since the $>_+$ and $>_-$ are both generated by the rules in E, each element $a_{i,j}$ must contain the timestamp n. Strong negation is only ever used with *holds*; and there can be no *occurs* in the chain since every rule in E with *occurs* in the head must be a fact. Similarly, there can be no *step_between* since the instances of rule (14) have been reduced to facts in E. By construction, *step*(n) is not a premise of any rule in E; and *not step*(t) is not even a premise of any rule in Γ (by property 1). Thus, *step* does not occur in the chain either. We allow other predicate symbols in Γ_b, but their rules do not contain *not* (nor predicates with rules that do). Hence, these symbols may not occur either, and we are left with *holds*(f, n) and $-$*holds*(f, n) for ground fluents f.

Next, we define a corresponding labelled path with nodes

$$x_{i,j} = \begin{cases} f & \text{if } a_{i,j} = \mathit{holds}(f, n), \\ -f & \text{if } a_{i,j} = -\mathit{holds}(f, n). \end{cases}$$

If $a_{i,n_i} >_- a_{i+1,0}$ because of an inertia rule or closed world assumption in E, rules (12), (13) and (15), then we include the edge $x_{i,n_i} \overset{!}{\Leftarrow} x_{i+1,0}$. It is easy to see that every other step $a_{i,j} >_s a_{i',j'}$ (where $s \in \{+, -\}$) must be due to a rule $R \in E$ which is the result of deleting zero or more premises from a ground causal rule. Thus, we may include the corresponding edge $x_{i,j} \overset{s}{\Leftarrow} x_{i',j'}$ in the path.

Since we now have a path with an infinite number of edges marked $-$ or !, it contains both $f \overset{!}{\Leftarrow} -f$ and $-f \overset{!}{\Leftarrow} f$ for some f by property 4. This means that E contains instances of both inertia rules for the pair (f, n), possibly replacing the negative inertia rule with the corresponding closed world assumption if $n = 0$. For the positive inertia rule instance to be included in E, we know that $I\restriction_n$ must contain *holds*(f, k) where $k = \max\{i \mid \mathit{step}(i) \in I\restriction_n\}$. Thus, $n > 0$ and $I\restriction_n$ must also contain $-$*holds*(f, k). Since this contradicts the fact that $I\restriction_n$ is sound, there is no such infinite decreasing chain. Thus, E is locally stratified. □

Assume that I and J are answer sets for Δ and that I is sound. In order to complete the proof of Theorem 3, we must show that $I = J$. From Lemma 1 we see that $I\restriction_j$ and $J\restriction_j$ are both answer sets for $\Delta\restriction_j$ for every j. Also, $I\restriction_j$ must be sound since it is a subset of I. Thus $I\restriction_j = J\restriction_j$ for every j by Lemma 2; and since every atom in I is a member of $I\restriction_j$ for some j (and J has the same property), we get $I = J$. This concludes the proof of the theorem.

Funding

This research was funded by the Research Council of Norway under project no. 282081 (MixStrEx) and no. 329062 (ASCERT).

References

1. Ericsson KA. An introduction to Cambridge Handbook of Expertise and Expert Performance: Its development, organization, and content. In Ericsson KA, Charness N, Feltovich PJ et al. (eds.) *The Cambridge Handbook of Expertise and Expert Performance*, chapter 1. Cambridge Univ. Press, 2006. pp. 3–20.
2. Hannay JE and Kikke Y. Structured crisis training with mixed reality simulations. In *Proc. 16th Int'l Conf. Information Systems for Crisis Response and Management (ISCRAM)*. pp. 1310–1319.
3. Papineau D. Philosophy of science. In Bunnin N and Tsui-James EP (eds.) *The Blackwell Companion to Philosophy*. Blackwell Publishing, 1996. pp. 286–317.
4. Zeigler BP, Praehofer H and Kim TG. *Theory of Modeling and Simulation*. 2nd ed. Academic Press, 2000.
5. Baral C and Gelfond M. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence*. Springer, 2000. pp. 257–279.
6. Gelfond M and Kahl Y. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.
7. Baral C, Gelfond G, Pontelli E et al. An action language for multi-agent domains. *Artificial Intelligence* 2022; 302: 103601.
8. Lifschitz V. *Answer set programming*. Springer Heidelberg, 2019.
9. Giunchiglia E, Lee J, Lifschitz V et al. Nonmonotonic causal theories. *Artificial Intelligence* 2004; 153(1): 49–104. Logical Formalizations and Commonsense Reasoning.
10. Makinson D. General patterns in nonmonotonic reasoning. In *Handbook of Logic in Artificial Intelligence Nad Logic Programming, Vol. Iii*. Oxford: Clarendon Press, 1994. pp. 35–110.
11. Schlechta K. *Coherent systems*. Elsevier, 2004.
12. Makinson D. A tale of five cities. In *David Makinson on Classical Methods for Non-Classical Problems*. Springer, 2014. pp. 19–32.
13. Gebser M, Kaminski R, Kaufmann B et al. Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning* 2012; 6(3): 1–238.
14. Stolpe A and Hannay JE. Quantifying means-end reasoning skills in simulation-based training: a logic-based approach. *Simulation* 2022; 98(10): 933–957.
15. Zeigler BP. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation* 1987; 49(5): 219–230.
16. Chow ACH and Zeigler BP. Parallel devs: A parallel, hierarchical, modular modeling formalism. In *Proceedings of Winter Simulation Conference*. IEEE, pp. 716–722.
17. Cho SM and Kim TG. Real-time devs simulation: Concurrent, time-selective execution of combined rt-devs model and interactive environment. In *SCSC-98*. pp. 410–415.
18. Vangheluwe H. The discrete event system specification (devs) formalism. *tech rep* 2001; .
19. Baral C. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
20. Pearce D. Equilibrium logic. *Annals of Mathematics and Artificial Intelligence* 2006; 47(1): 3.
21. Gebser M, Kaminski R, Kaufmann B et al. *Answer Set Solving in Practice*. Morgan & Claypool Publishers, 2012.
22. Cristiá M. Formalizing the semantics of modular devs models with temporal logic. In *Proceeedings of 7me Conférence Internationale de Modélisation, Optimisation et Simulation des Systemes: Communication, Coopéeration et Coordination (MOSIM 08)*.
23. Arias J, Carro M, Salazar E et al. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 2018; 18(3-4): 337–354.
24. IEEE Standards Association. *1516-2010 – IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)*, 2010.
25. Simulation Interoperability Standards Organization. *SISO-STD-003-2006 – Standard for Base Object Model (BOM) Template Specification*, 2006.
26. Mojtahed V, Andersson B, Kabilan V et al. BOM++, a semantically enriched BOM. In *Proc. 2008 Spring Simulation Interoperability Workshop (SIW)*. Simulation Interoperability Standards Organization.
27. De Nicola A, Melchiori M and Villani ML. Creative design of emergency management scenarios driven by semantics: An application to smart cities. *Information Systems* 2019;

81: 21–48.

28. Steel J, Iannella R and Lam HP. Using ontologies for decision support in resource messaging. In *Proc. 5th Int'l Conf. Information Systems for Crisis Response and Management (ISCRAM)*. pp. 189–196.
29. Bénaben F, Hanachi C, Lauras M et al. A metamodel and its ontology to guide crisis characterization and its collaborative management. In *Proc. 5th Int'l Conf. Information Systems for Crisis Response and Management (ISCRAM)*. pp. 189–196.
30. Singapogu SS, Gupton K and Schade U. The role of ontology in C2SIM. In *Proc. 21st International Command and Control Research and Technology Symposium (ICCRTS 2016)*. The International Command and Control Institute. Paper 12.
31. Simulation Interoperability Standards Organization. *The Command and Control Systems – Simulation Systems Interoperation (C2SIM) Product Development Group (PDG) and Product Support Group (PSG)*, 2019. Accessed December 19, 2019.
32. van den Berg TW, Huiskamp W, RobertSiegfried et al. 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*.
33. Kasım B, Çavdar AB, Nacar MA et al. Modeling and simulation as a service for joint military space operations simulation. *The Journal of Defense Modeling and Simulation* 2021; 18(1): 29–38.
34. Hannay JE, van den Berg T, Gallant S et al. Modeling and Simulation as a Service infrastructure capabilities for discovery, composition and execution of simulation services. *J Defense Modeling and Simulation: Applications, Methodology, Technology* 2020; 1(4): 5–28.
35. Hannay JE and van den Berg TW. 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)*. NATO Science and Technology Organization. Paper 13.
36. Mentré D, Marché C, Filliâtre JC et al. Discharging proof obligations from Atelier B using multiple automated provers. In Derrick J, Fitzgerald J, Gnesi S et al. (eds.) *Abstract State Machines, Alloy, B, VDM, and Z*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 238–251.
37. Abrial JR. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
38. Morris K, Snook C, Hoang TS et al. Formal verification and validation of run-to-completion style state charts using Event-B. *Innovations in Systems and Software Engineering* 2022; 18(4): 523–541.
39. Gao Y, Zhang Y, Zhou X et al. Overview of simulation architectures supporting live virtual constructive (lvc) integrated training. In *2021 6th International Conference on Control, Robotics and Cybernetics (CRC)*. IEEE, pp. 333–338.
40. Hodicky J, Prochazka D and Prochazka J. Automation in experimentation with constructive simulation. In Mazal J (ed.) *Modelling and Simulation for Autonomous Systems*. Cham: Springer International Publishing, pp. 566–576.
41. Grigoryan G, Etemadidavan S and Collins AJ. Computerized agents versus human agents in finding core coalition in glove games. *SIMULATION* 2022; 98(9): 807–821.
42. Løvlid RA, Bruvoll S, Brathen K et al. Modeling the behavior of a hierarchy of command agents with context-based reasoning. *J Defense Modeling and Simulation: Applications, Methodology, Technology* 2018; 15(4): 369–381.
43. Zakaria N. Action network: a probabilistic graphical model for social simulation. *SIMULATION* 2022; 98(4): 335–346.
44. Simulation Interoperability Standards Organization. *SISO-STD-011-2014 – Standard for Coalition Battle Management Language (C-BML) Phase 1, Version 1.0*, 2014.
45. Angilica D, Ianni G and Pacenza F. Tight integration of rule-based tools in game development. In Alviano M, Greco G and Scarcello F (eds.) *AI*IA 2019 – Advances in Artificial Intelligence*. Cham: Springer International Publishing, pp. 3–17.
46. Calimeri F, Fuscà D and Germano S. Fostering the use of declarative formalisms for real-world applications: The embasp framework. *New Generation Computing* 2019; 37(1).
47. Azpiazu RA. clingo-cs, 2021. URL `https://github.com/NEKERAFA/clingo-cs`. Original-date: 2020-01-15T16:57:14Z.
48. Gelfond M and Lifschitz V. The stable model semantics for logic programming. In Kowalski R, Bowen and Kenneth (eds.) *Proceedings of International Logic Programming Conference and Symposium*. MIT Press, pp. 1070–1080.
49. Gelfond M and Lifschitz V. Classical negation in logic programs and disjunctive databases. *New Generation*

Computing 1991; 9: 365–385.
50. Kowalski R. Algorithm = logic + control. *Commun ACM* 1979; 22(7): 424–436.
51. Kowalski R. *Computational logic and human thinking: how to be artificially intelligent*. Cambridge University Press, 2011.
52. Giunchiglia E, Lee J, Lifschitz V et al. Nonmonotonic causal theories. *Artificial Intelligence* 2004; 153(1–2): 49–104.
53. Mueller ET. Chapter 17 event calculus. In van Harmelen F, Lifschitz V and Porter B (eds.) *Handbook of Knowledge Representation*, *Foundations of Artificial Intelligence*, volume 3. Elsevier, 2008. pp. 671–708.
54. Oberschelp A. Order sorted predicate logic. In Bläsius KH, Hedtstück U and Rollinger CR (eds.) *Sorts and Types in Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 7–17.
55. Mueller ET. *Commonsense reasoning: an event calculus based approach*. Morgan Kaufmann, 2014.
56. Reiter R. On closed world data bases. In Webber BL and Nilsson NJ (eds.) *Readings in Artificial Intelligence*. Morgan Kaufmann, 1981. pp. 119–140.
57. Kaufmann B, Leone N, Perri S et al. Grounding and solving in answer set programming. *AI magazine* 2016; 37(3): 25–32.
58. Gebser M, Schaub T and Thiele S. Gringo: A new grounder for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 266–271.
59. Breien FS and Wasson B. Narrative categorization in digital game-based learning: Engagement, motivation & learning. *British Journal of Educational Technology* 2021; 52(1): 91–111.
60. Subhash S and Cudney EA. Gamified learning in higher education: A systematic review of the literature. *Computers in Human Behavior* 2018; 87: 192–206.
61. Campos N, Nogal M, Caliz C et al. Simulation-based education involving online and on-campus models in different european universities. *International Journal of Educational Technology in Higher Education* 2020; 17(1): 8.
62. Gebser M, Kaminski R, Kaufmann B et al. Clingo = asp + control: Preliminary report, 2014.

Author biographies

Audun Stolpe is a senior researcher at the Norwegian Computing Center. His current research explores applications of declarative knowledge representation and logic pro- gramming for simulation based training, information security, decision support and planning. A doctor of philosophy his general research interests fall in the intersection of pure and applied logic, philosophy and information science.

Ivar Rummelhoff is a senior research scientist at the Norwegian Computing Center, mostly working within the area of digital transformation. Ivar has a PhD in mathematical logic.

Jo Erskine Hannay is a senior researcher at the Center for Effective Digitalization of the Public Sector at Simula Metropolitan Center for Digital Engineering. He conducts research in simulation-based training with a particular focus on training judgement and decision making in crisis management and in health care collaboration. He also conducts research on benefits estimation and evaluation of IT systems and services, both from the perspective of systems development and from the perspective of stakeholders affected by such systems and services.