Combining Event Calculus and Description Logic Reasoning via Logic Programming

Peter Baumgartner

Data61/CSIRO and The Australian National University, Canberra, Australia
Peter.Baumgartner@data61.csiro.au

Abstract. The paper introduces a knowledge representation language that combines Kowalski's event calculus with description logic in a logic programming framework. The purpose is to provide the user with an expressive language for modelling and analysing systems that evolve over time. The description logic component is intended for modelling structural properties, the event calculus for actions and their consequences, and the logic programming rules for their integration and other aspects, such as diagnosis. By means of an elaborated example, the paper demonstrate the interplay of these three components for computing possible models as plausible explanations of the current state of the modelled system. The approach is prototypically implemented in our logic programming system Fusemate. The paper first extends Fusemate's rule language with a weakly DL-safe interface to the description logic $\mathcal{ALCIF}$ (which is implemented in Fusemate itself). It then embeds a suitable version of the event calculus, and provides rules as the "glue" between these components.

1 Introduction

This paper presents an expressive logical language for modelling systems that evolve over time. The language is intended for model computation: given a history of events until "now", what are the system states at these times, in particularly "now", expressed as logical models. This is a useful reasoning service in application areas with only partially observed events or incomplete domain knowledge. By making informed guesses and including its consequences, the models are meant to provide plausible explanations for helping understand the current issues, if any, as a basis for further decision making.

For example, transport companies usually do not keep detailed records of what goods went on what vehicle for a transport on a particular day. Speculating the whereabouts of a missing item can be informed by taking known locations of other goods of the same batch on that day into account; problems observed with goods on delivery site, e.g., low quality of fresh goods, may or may not be related to the transport conditions, and playing through different scenarios may lead to plausible explanations while eliminating others (truck cooling problems? tampering?).

There are numerous approaches for modelling and analysing systems that evolve over time. They are often subsumed under the terms of stream processing, complex event recognition, and situational awareness, temporal verification among others, see [1,14,3,4,5] for some logic-based methods. Symbolic event recognition, for

instance, accepts as input a stream of time-stamped low-level events and identifies high-level events — collections of events that satisfy some pattern [1]. See [41] for a recent sophisticated event calculus. Other approaches utilize description logics in a temporalized setting of ontology-based data access (OBDA) [34]. For instance, [33] describes a method for streaming data into a sequence of ABoxes, which can be queried in an SQL-like language with respect to a given ontology.

The knowledge representation language put forward in this paper combines Kowalski's event calculus (EC) with description logics (DL) in a logic programming framework. The rationale is, DLs have a long history of developments for representing structured domain knowledge and for offering reliable (decidable) reasoning services. The EC provides a structured way of representing actions and their effects, represented as fluents that may change their truth value over time. For the intended model computation applications mentioned above, the EC makes it easy to take snapshots of the fluents at any time. The full system state at a chosen time then is derived from the fluent snapshot and DL reasoning. The logic programming rules orchestrate their integration and serve other purposes, such as diagnosis.

This paper is meant as an initial investigation into integrating DL into the EC. Technically, the developments build on the Fusemate logic programming language and system [11,12]. In brief, Fusemate is a logic-programming system for bottom-up computation of possible models of disjunctive logic programs [37,38]. A Fusemate logic program consists of (typically) non-ground if-then rules with stratified default negation in the body [35]. Fusemate is tightly integrated with its host programming language Scala [39] in terms of callability from/to Scala and data structures.

Fusemate was introduced in [11] with the same motivation as here. The underlying calculus features a correspondingly suitable notion of stratification based on the time line for making the calculus effective. It also features a simple belief revision operator that is particularly useful in this context. This operator enables amending computed models retrospectively, based on new information coming in over time.

The follow-up paper [12] introduced a weaker form of stratification. This "stratification by time and predicates" (SBTP) extends lexicographically the above stratification by time with standard stratification in terms of the call-graph of the program. That paper also introduced novel aggregation and comprehension operators and demonstrated the usefulness in combination with SBTP with an application to description logic reasoning. More precisely, the paper [12]) demonstrates how to transform an $\mathcal{ALCIF}$[1] knowledge base (a TBox and an Box) into a set of Fusemate rules and facts that is satisfiable if and only if the knowledge base is $\mathcal{ALCIF}$-satisfiable. All of that is used in this paper.

Paper contributions. This paper builds on the Fusemate developments summarized above and extends it in the following ways:

1. Integration of the description logic reasoner of [12] as a subroutine callable from Fusemate rules. This is a hybrid combination method that imposes no restrictions on the DL language (such as TBox acyclicity).

[1] $\mathcal{ALCIF}$ is the well-known description logic $\mathcal{ALC}$ extended with inverse roles and functional roles. See [2] for background on description logics.

While integrating description logic into a rule language has been done before, Fusemate specifics (e.g., stratification) enable a more expressive interface than other approaches. Details in Section 4.
2. Formulating a version of the event calculus [22] that fits Fusemate's model computation and notion of stratification. Details in Section 5,
3. Integrating DL and EC by means of rules, and utilizing rules for KR aspects not covered by either. Details in particular in Section 6
4. Providing an elaborated example for demonstrating the capabilities of the integrated EC/DL/rules language and its implementation. Both are available at `https://bitbucket.csiro.au/users/bau050/repos/fusemate/`.

To the best of my knowledge, a combination of DL with EC has not been considered before. Given the long history of applying DL reasoning (also) for time evolving systems, I find this surprising. From that perspective, the main contribution of this paper is to fill the gap and to argue that the proposed combination makes sense.

There is work is on integrating DLs into the situation calculus (SitCalc) or similar methods, though [16,9,8,10]. SitCalc [25] is a first-order logic formalism for specifying state transitions in terms of pre- and post-conditions of actions and this way is related to EC. SitCalc is mostly used for planning and related applications that require reachability reasoning for state transitions. Indeed, the papers [9] and [16] investigate reasoning tasks (executability and projection, ABox updates) that are relevant in that context. Both approaches are restricted to acyclic TBoxes. In [10], actions are specified as sets of conditional effects, where conditions are based on epistemic queries over the knowledge base (TBox and ABox), and effects are expressed in terms of new ABoxes. The paper investigates verification of temporal properties. In difference to the EC, none of these approaches supports a *quantitative* notion of time.

2 Stratified Logic Programs and Model Computation

For the purpose of this paper, a brief summary of the Fusemate logic programming system and its rule language is sufficient. It is based on the earlier papers [11,12]. Terms and atoms of a given first-order signature are defined as usual. Let $var(z)$ denote the set of variables occurring in an such expression z. We say that z is *ground* if $var(z) = \emptyset$. We write $z\sigma$ for applying a substitution σ to z. The domain of σ is denoted by $dom(\sigma)$. A substitution γ is a *grounding substitution for z* iff $dom(\gamma) = var(z)$ and $z\gamma$ is ground. In this case we simply say that *γ is for z*.

Let $\mathbb{T}$ be a countably infinite discrete set of *time points* equipped with a total strict ordering $<$ ("earlier than"), e.g., the integers. Assume that the time points, comparison operators $=$ and $\leq$, and a successor time function $+1$ are part of the signature and interpreted in the intended way. A *time term* is a (possibly non-ground) term over the sub-signature $\mathbb{T} \cup \{+1\}$. In this paper, $\mathbb{T}$ is the integers.

The signature may contain other "built-in" predicate and function symbols for predefined types such as strings, arithmetic data types, sets, etc. We only informally assume that all terms are built in a well-sorted way and that built-in operators over ground terms can be evaluated effectively.

An *ordinary atom (with time term t)* is of the form $p(t, t_1, \ldots, t_n)$ where p is an ordinary predicate (i.e., neither a time predicate nor built-in), t is a time term and $t_1, \ldots, t_n$ terms. A *(Fusemate) rule* is an implication written in Prolog-like syntax as

$$H \text{ :- } b_1, \ldots, b_k, \textbf{not}\, \vec{b}_{k+1}, \ldots, \textbf{not}\, \vec{b}_n \ . \tag{1}$$

In (1), a rule *head* H is either (a) a disjunction $h_1 \vee \cdots \vee h_m$ of ordinary atoms, for some $m \geq 1$, or (b) the expression **fail**.[2] In case (a) the rule is *ordinary* and in case (b) it is a *fail rule*. A rule *body* B, the part to the right of :–, is defined by mutual recursion as follows. A *positive body literal* is one of the following: (a) an ordinary atom, (b) a *comprehension atom (with time term x)* of the form $p(x \circ t, t_1, \ldots, t_n)$ **sth** B, where x is a variable, $\circ \in \{<, \leq, >, \geq\}$ and B is a body, (c) a built-in call , i.e., an atom with a built-in predicate symbol, or (d) a *special form* $\textbf{let}(x, t)$, $\textbf{choose}(x, ts)$, $\textbf{match}(t, s)$ or $\textbf{collect}(x, t\ \textbf{sth}\ B)$ where x is a variable, s, t are terms, ts is a list of terms, and B is a body. A *positive body* is a list $\vec{b} = b_1, \ldots, b_k$ of positive body literals with $k \geq 0$. If $k = 0$ then $\vec{b}$ is *empty* otherwise it is *non-empty*. Semantically, the list represents a conjunction, but operationally it is worked-off from left to right, otherwise comprehension and the special forms would not make much practical sense. See [12] for a formal definition. A *negative body literal* is an expression of the form $\textbf{not}\, \vec{b}$, where $\vec{b}$ is a non-empty positive body. A *body* is a list $B = b_1, \ldots, b_k, \textbf{not}\, \vec{b}_{k+1}, \ldots, \textbf{not}\, \vec{b}_n$ comprised of a (possibly empty) positive body and (possibly zero) negative body literals. It is *variable free* if $var(b_1, \ldots, b_k) = \emptyset$.

Let r be a rule (1). We say that r is *range-restricted* iff $var(H) \subseteq var(\vec{b})$. Compared to the usual notion of range-restrictedness [28], Fusemate rules may contain *extra variables* in negative body literals. For example, $\mathsf{p}(t, x)$:- $\mathsf{q}(t, x), \textbf{not}(s < t, \mathsf{r}(s, x, y))$ is range-restricted in our sense with extra variables s and y. The extra variables are implicitly existentially quantified within the **not** expression and are those variables that are not bound by body matcher computation from the preceding body literals. The example corresponds to the formula $\mathsf{q}(t, x) \wedge \neg \exists s, y.(s < t \wedge \mathsf{r}(s, x, y)) \rightarrow \mathsf{p}(t, x)$. Similarly, the conditions B in the "**sth** B" expressions may have extra variables, which are also locally existentially quantified. Semantically and operationally the extra variables will cause no problems thanks to stratification (see below).

Some examples below employ a classical negation operator **NEG** ("strong negation") which can be applied to ordinary atoms in the body or the head. Fusemate implements the usual semantic [19] which amounts to adding the rules $\textbf{FAIL}$:- $p(\vec{x}), \textbf{NEG}(p(\vec{x}))$ for every ordinary predicate symbol p.

Model Computation. Fusemate programs, or just *programs*, are sets of range-restricted rules. Starting from a given set of ground atoms, the *given facts*, or *EDB*, the Fusemate system computes possible models by bottom-up fixpoint computation and dynamic grounding and splitting the program rules in the style of hyper tableaux [13] (see [11] for details). At every stage in the computation, each branch in the tableau represents an interpretation – a set of atoms – that is to be completed into a model or abandoned

[2] This definition of head is actually simplified as Fusemate offers an additional head operator for belief revision, see[11]. This is ignored here.

eventually. There is always a selected branch referred to as the *current interpretation I* and *current time time*. Rules are evaluated by processing their bodies from left to right, thereby grounding the positive body literals by matching the ordinary and comprehension atoms to atoms from I. At each step, the substitution aggregated so far, the *body matcher*, is applied to the current body literal prior to processing it. See [12] for details.

In addition to range-restrictedness every program must be stratified. The crucial property of (any form of) stratification is that a true negative body literal remains true when the model computation adds new consequences to the current (partial) model candidate. This eliminates the need for guessing whether a negative body literal is true or false in the final model. The *standard* notion of stratification ("by predicates") means that the call graph of the program contains no cycles going through negative body literals. The edges of this call graph are the "depends on" relation between predicate symbols such that p positively (negatively) depends on q if there is a rule with a p-atom in its head and a q-atom in its positive (negative) body. For ordinary heads, all head predicates are defined to depend positively on each other. Every strongly connected component of the call graph is called a stratum, and in predicate stratified programs negative body literals can occur only in strata lower than the head stratum.

For better practical usability, Fusemate employs *stratification by time and by predicates (SBTP)*. With SBPT, (ground) atoms are first compared by their timestamps, or else, if these are equal, by their strata. A ground rule is SBTP if its ordinary body literals (within negative body literals) are below (strictly below, respectively) than its head literals(s) wrt. this ordering. Comprehension atoms and special forms must be stratified in the same way. The time component of SBTP can be enforced by adding constraints, and the stratum component is computed automatically by Tarjan's algorithm (in some cases the user needs to annotate hints). For example, the rule $\mathsf{p}(\mathit{time}, x)$:- $\mathsf{q}(\mathit{time}, x), \mathbf{not}(\mathsf{r}(t, y), t \leq \mathit{time})$ is SBTP if r belongs to a strictly lower stratum than p, and $\mathsf{p}(\mathit{time}, x)$:- $\mathsf{q}(\mathit{time}, x), \mathbf{not}(\mathsf{r}(t, y), t < \mathit{time})$ is SBTP even if r belongs to the same stratum as p. See [12] for details on SBTP.

Fusemate is implemented by shallow embedding in the Scala programming language [39]. Rules are translated into Scala source code by a compiler plugin. All data structures, including atoms and interpretations and those manipulated by a program are Scala, Scala is callable from within rules, and Scala is Fusemate's extension ("scripting") language.

3 Running Example

This running example in the section helps to demonstrate the interplay of the three components of the combined description logic, rules, and event calculus language.

We consider a highly simplified transport scenario. Boxes containing goods are loaded onto a truck, moved to a destination, and unloaded again. The boxes can contain perishable goods that require cooling, fruits, or non-perishable goods, toys. Boxes of the former kind (and only those) can be equipped with temperature sensors and provide a temperature value, which is classified as low (unproblematic) or high (problematic). At the destination a box with a high temperature arrives and the problem is to advise the

user with plausible explanations for that. The modelling in the example supports that "the box has been tampered with" or "the cooling of the truck broke down".

The explanations will computed as (logical) models of the domain model, the concrete objects in the scenario, and the given events. The domain model consists of an ontology and if-then rules for diagnosing high temperatures. The concrete objects are boxes, and events are timestamped loading/unloading actions and temperature sensor readings. The ontology is given as a TBox in the description logic $\mathcal{ALC}$ extended with functional roles (left); the ABox consists of individuals for temperature classes (middle) and box individuals of varying attributes (right):

$\mathsf{Box} \sqsubseteq \forall\, \mathsf{temp}.\mathsf{TempClass}$	$\mathsf{Low} : \mathsf{TempClass}$	$\mathsf{Box}_0 : \mathsf{FruitBox}$
$\mathsf{FruitBox} \sqsubseteq \exists\, \mathsf{temp}.\mathsf{TempClass}$	$\mathsf{High} : \mathsf{TempClass}$	$\mathsf{Box}_1 : \mathsf{FruitBox}$
$\mathsf{ToyBox} \sqsubseteq \neg\exists\, \mathsf{temp}.\mathsf{TempClass}$		$\mathsf{Box}_2 : \mathsf{Box}$
$\mathsf{FruitBox} \sqsubseteq \mathsf{Box}$		$\mathsf{Box}_3 : \mathsf{ToyBox}$
$\mathsf{ToyBox} \sqsubseteq \mathsf{Box}$		$\mathsf{Box}_4 : \mathsf{Box} \sqcap \forall\, \mathsf{temp}.\neg\mathsf{TempClass}$
temp is a functional role		$\mathsf{Box}_5 : \mathsf{Box} \sqcap \exists\, \mathsf{temp}.\mathsf{TempClass}$

The domain model also includes actions and their effects, event calculus style. The actions are "loading", "unloading", and the only effect is "on". As an event calculus, effects are fluents that are "initiated" by an action and will hold until "terminated" by a later action. The actions are:[3]

1. A box loading action initiates the box to be on the truck.
2. An unloading action terminates every box to be on the truck.

The diagnosis rules for explanations can be stated informally as follows. Suppose a given subset of the boxes $\{\mathsf{Box}_0, \ldots, \mathsf{Box}_5\}$ have been unloaded at the destination.

1. If there is no unloaded box with known high temperature then the status is OK.
2. If some unloaded box has a known high temperature then this box has been tampered with or the truck cooling is broken.
3. If some unloaded box has a known low temperature then the truck cooling is not broken (because a broken cooling affects all boxes).
4. Suppose that all unloaded known fruit boxes can consistently be assumed to have high temperature. Then box tampering can be excluded (because broken cooling is the more likely explanation).

The formalization of these rules below distinguishes between absent, unknown and known box temperature attributes. This yields different explanations depending on the actual events and what is known about the unloaded boxes at unloading time.

One scenario, for instance, unfold as follows:

Time	10	20	30	40	50
Action	Load Box_0	Load Box_2	Load Box_3		Unload
	Load Box_1		Load Box_4		
Sensor	$\mathsf{Box}_0 : -10°$	$\mathsf{Box}_2 : 10°$	$\mathsf{Box}_0 : 2°$	$\mathsf{Box}_0 : 20°$	

The diagnosis is "the cooling is broken". The rest of the paper explains the underlying modelling.

[3] I use the term "action" where usual event calculus terminology is "event". Here the term "event" is wider and means anything happening, e.g., also temperature sensor readings.

4 Description Logic Interface

The recent paper [12] (also) describes how Fusemate can be utilized for description logic reasoning. This section makes that reasoner callable from within Fusemate rules. This Fusemate-within-Fusemate approach was relatively easy to implement and has its advantages, but other DL reasoners could be coupled as well. As for the DL requirements, concept formation must be closed under conjunction and negation ($\mathcal{ALC}$), so that entailment is reducible to satisfiability.

A brief summary of Fusemate as a DL reasoner is useful. A DL knowledge base KB consists of an ABox and TBox. An ABox is a set of concept and role assertions. A TBox is a set of GCIs (general concept inclusions). As usual, KB-satisfiability is assumed to be satisfiable. (See [2] for background on DLs.) Fusemate currently implements satisfiability checking for $\mathcal{ALCIF}$, which is $\mathcal{ALC}$ extended with inverse roles and functional roles. The implementation is by translation of the KB into Fusemate rules and facts. The starting point is the FOL version of the given KB, where concept names become unary predicates, role names become binary predicates, and GCIs are translated into implications. The implications then are turned into rules over concept and role atoms by expansion laws and moving negated and universally quantified formulas into the head via complementation. The most problematic form is existential quantification in the head, which would cause unbounded Skolem terms in derivations, and must be treated differently. Several such translations were proposed in the literature, see e.g. [31,26,21,15]. The Fusemate implementation only needs to provide a generic rule library for expanding quantified formulas under blocking conditions, Boolean expansion into CNF, and some more auxiliary concepts. With that, a given KB satisfiability question can be directly handed off to the Fusemate reasoner as an equi-satisfiable problem without the need for an explicitly implemented DL proof procedure.

Knowledge base representation. The Fusemate representation of a KB works with reifed concept and role assertions. There are two dedicated predicate symbols for that, IsA/2 and HasA/3, respectively. Corresponding *DL-atom*s are of the form $\mathsf{IsA}(t, c)$ and $\mathsf{HasA}(c_1, r, c_2)$ where c, c_1, c_2 stand for concept and r for role expressions. For example, the translation of the TBox in Section 3 into Fusemate rules is as follows (notice there is no *time* parameter in these rules, these are added automatically for convenience):

```
1 IsA(x, Forall(Temp, TempClass)) :– IsA(x, Box)
2 IsA(x, Exists(Temp, TempClass)) :– IsA(x, FruitBox)
3 FAIL :– IsA(x, ToyBox), HasA(x, Temp, y), IsA(y, TempClass)
4 IsA(x, Box) :– IsA(x, FruitBox)
5 IsA(x, Box) :– IsA(x, ToyBox)
```

ABox assertions are represented analogously. For example, the Box_5 assertion in Section 3 becomes IsA(Box(5),And(Box,Exists(Temp,TempClass))). However, this is only the DL reasoner internal representation. For analysing systems that change over time it is more useful to support timed ABoxes, as, of course, data will change over time. To this end, Fusemate predefines *timed DL-atoms* of the form $\mathsf{IsAAt}(time, t, c)$ and $\mathsf{HasAAt}(time, c_1, r, c_2)$ which can be used in rules and facts. Below we will see how they are translated into their untimed counterparts as part of the interface to the DL reasoner.

In the example, for every time point $t = 10, 20, \ldots, 50$ the given facts include the timed DL-atom IsAAt(t,Box(5),And(Box,Exists(Temp,TempClass))), and likewise for the other boxes, as well as IsAAt(t, Low, TempClass) and IsAAt(t, High, TempClass). In fact, as will be seen in Section 5, these facts will be derived from fluents by event calculus rules, so that they need to be given explicitly only once.

DL-reasoner interface. Assume that the concept names and role names are disjoint from the predicate and function symbols of the rule language. However, the (ground) terms of the rule language are shared, they are also individuals of the DL language and called *known* individuals in that context. The interface, proper, is given by the following *DL-call* special forms, where $\vec{c}$ is a list of DL-atoms.

$tbox \mid= \vec{c}$	$\mathsf{DLISSAT}(tbox)$	$\mathsf{DLISUNSAT}(tbox)$
$(abox, tbox) \mid= \vec{c}$	$\mathsf{DLISSAT}(abox, tbox)$	$\mathsf{DLISUNSAT}(abox, tbox)$

As a special form (cf. Section 2) is a positive body literal and can appear in nearly any position, including within negative body literals (where stratification must take implicitly referred timed DL-atoms into account). The *DL-entailment* form in the left column is the most interesting one. With respect to variables, DL-entailments are treated like negative body literals and may contain extra, existentially quantified variables in $\vec{c}$. Let d be a DL-entailment call occuring in a rule r. Let $\vec{c}$ be its DL-atoms list with a set $\mathbf{x}$ of extra variables. We say that *d is admissible in r* if for every ground substitution γ with $dom(\gamma) = var(\vec{c}) \setminus \mathbf{x}$, the formula $\exists \mathbf{x} \bigwedge \vec{c}\gamma$ is equivalent to some conjunction of ABox assertions wrt. the first-order logic semantics of DL. A rule is admissible if every of its DL-entailments is admissible. We consider only programs with admissible rules.

For the purpose of this paper this somewhat vague characterization should be enough. The intent is to make sure that a DL-entailment represents an (entailment) problem that can be decided by the DL reasoner. During rule evaluation, a DL-entailment $(abox, tbox) \mid= \vec{c}$ reached with a body matcher γ (see Section 2) represents the question if the KB $(abox, tbox)$ as a first-order logic formula entails $\exists \mathbf{x} \bigwedge \vec{c}\gamma$. By admissibility, this question can be answered by checking $r_i : (C_i, D_i) \in abox$ for every role assertion $r_i : (C_i, D_i)$ and checking DL unsatisfiability of $(abox \cup \{a_i : \neg C_i\}, tbox)$ for every concept assertion $a_i : C_i$ induced by admissibility.[4]

In the second form of DL-entailments the ABox is not specified. In this case, the ABox is extracted from the current interpretation I as $abox(time) = \{\mathsf{IsA}(x, c) \mid \mathsf{IsAAt}(t, x, c) \in I \text{ and } t = time\} \cup \{\mathsf{HasA}(x, r, y) \mid \mathsf{HasAAt}(t, x, r, y) \in I \text{ and } t = time\}$. The concrete syntax for that is I.aboxAt(time).[5]

Example 1. Consider the rule

```
1 TempBox(time, box) :–
2         IsAAt(time, box, Box),
3         tbox |= HasA(box, Temp, temp), IsA(temp, TempClass)
4         // equivalently: (I.aboxAt(time),tbox) |= HasA(box, Temp, temp), IsA(temp, TempClass)
```

[4] Actually, the Fusemate DL-reasoner can be queried more efficiently without detours.

[5] Observe that rules can access the current interpretation I, which is unusual for logic programming systems. See again [12] for a discussion of such features.

The intention of this rule is to record that a box existing at time *time* has a Temp attribute. Suppose tbox is the TBox from above, and the current time time is 10. Suppose the body atom is already instantiated to IsAAt(10, Box(2), Box)). By admissibility and with the extra variable *t* being existentially quantified, the DL-entailment call on line 3 evaluates to true if and only if the KB $(abox(10) \cup \{\mathsf{Box}_2 : \neg\exists\ \mathsf{temp.TempClass}\}, tbox)$ is unsatisfiable. The result will be false as Box_2 does not provably have a temp attribute. □

A good example for the KB unsatisfiability checking special form DLISUNSAT is the rule **FAIL** :– Now(time), DLISUNSAT(tbox). This rule abandons a current model candidate if its ABox is inconsistent with the TBox.

Implicit objects and properties. One would intuitively expect that the rule in Example 1 is applicable to Box_5. This, however, is not the case. The reason is that the body literal IsAAt(time, box, Box) does not syntactically match to Box_5's ABox declaration IsAAt(10,Box(5),And(Box,Exists(Temp,TempClass))). The problem is that IsAAt(10, Box(5), Box) is a concept assertion that is only *entailed* but not explicitly given. The same problem arises with Box_1 which requires the TBox to derive that Box_1 is a Box. The alternative TempBox(time, box) :– tbox |= IsA(box, Box), HasA(box, Temp, t), IsA(t, TempClass) does not solve the problem as it leaves a free variable box in the head. A correct solution is a rule that materializes the is-a-Box relation, like so:

```
1 IsAAt(time, x, Box) :–
2         IsAAt(time, x, _),
3         tbox |= IsA(x, Box)
```

From these two rules, the given facts and tbox, Fusemate will derive TempBox(10, Box(0)), TempBox(10, Box(1)), and TempBox(10, Box(5)) as expected.

Notice that in this rule the head literal is a timed DL-atom and, hence, can extend the current ABox. Thanks to monotonicity of first-order logic entailment this is not a problem, neither semantically nor operationally. In particular, notice that the extra variables in DL-entailment cannot "escape" and only already known individuals can instantiate the head. Moreover, the definition of possible model semantics can remain the same, only the definition of rule satisfaction needs to take the semantics of DL-calls into account.

Example 2. Consider the following variation of the rule in Example 1. The **AND** in the head is a shorthand for two rules with these head literals.

```
1 // Left head: box has known temperature; right head: materialization of HasAAt
2 KnownTempBox(time, box) AND HasAAt(time, box, Temp, temp) :–
3         IsAAt(time, box, Box),
4         CHOOSE(ttemp, List(Low, High)),
5         tbox |= HasA(box, Temp, temp), IsA(temp, TempClass)
```

The **CHOOSE** special form binds some element from {Low, High} to the variable temp. Suppose the DL-entailment call on line 4 is reached, say, for Box(1) at time 10 and in the Low case. Then both checks $(\mathsf{Temp} : (\mathsf{Box}(1), \mathsf{Low})) \in abox(10)$ and KB unsatisfiability of $(abox(10) \cup \{\mathsf{Low} : \neg\mathsf{TempClass}\}, tbox)$ fail. □

Examples 1 and 2 demonstrate how rules can help distinguish between unknown and known attribute values, respectively. Using default negation it is easy to write a rule that identifies the remaining case, when an individual does not have an attribute value at all.

Related work. Rules that allow extra variables in DL calls and that may have heads with DL atoms are called weakly DL-safe rules in DL+log [36]. DL+log is among the most expressive languages that combines rules with ontologies. Unlike DL+log, Fusemate allows DL calls within default negation, for example:

```
1 ColdBox(time, box) :–
2         IsAAt(time, x, Box),
3         NOT (t < time, (I.aboxAt(t), tbox) |= IsA(x, Box), HasA(x, Temp, High))
```

According to this rule, a box is a ColdBox at a given time if it never provably had a High temperature in the past.

Most other hybrid languages, like the one in [30] and dl+Programs [18] do not allow DL atoms in the head. Others do not allow extra variables in DL calls. See [17] for an in-depth overview of rule/DL combinations.

5 Event Calculus Embedding

The event calculus (EC) is a logical language for representing and reasoning about actions and their effects [22,40]. At its core, effects are fluents, i.e., statements whose truth value can change over time, and the event calculus provides a framework for specifying the effects of actions in terms of initiating or terminating fluents to hold.

Many versions of the EC exists, see [29] for a start. The approach below makes do with a basic version that is inspired by the discrete event calculus in [32] with integer time. The event calculus of [32] is operationalized by translation to propositional SAT. Its implementation in the "decreasoner" is geared for efficiency and can be used to solve planning and diagnosis tasks, among others. The version below is tailored for the model computation tasks mentioned in the introduction, where a fixed sequence of events at given timepoints can be supposed.[6]

With Fusemate implementing a minimal model semantics and with default negation available, there is no need for circumscription. For instance, frame axioms are stratified (by time) automatically and they work as intended without further ado. This is not a new invention and related answer set programming encodings of the event calculus have been proposed before, e.g. [23]. But, as said earlier, the main focus here is the integration with DL, which has not been done before.

Domain independent axioms. These are the domain-independent EC axioms here:

```
1 // DL assertions are fluents:
2 case class IsA(x: Individual, c: Concept) extends Fluent
3 case class HasA(x: Individual, r: Role, y: Individual) extends Fluent
```

[6] Actually, events can be inserted in retrospect using Fusemate's revision operator, restarting the model computation from there. The paper [11] already has a "supply-chain" example for that.

```
5  Initiated(time+1, f) :– Happens(time, a), Initiates(time, a, f) // H1
6  Terminated(time+1, f) :– Happens(time, a), Terminates(time, a, f) // H2
7  StronglyTerminated(time+1, f) :– Happens(time, a), StronglyTerminates(time, a, f) // H3
8  Terminated(time, f) :– StronglyTerminated(time, f) // H4

10 HoldsAt(time, f) :– Initiated(time, f), NOT Terminated(time, f) // EC3
11 NEG(HoldsAt(time, f)) :– StronglyTerminated(time, f), NOT Initiated(time, f) // EC4

13 HoldsAt(time, f) :– Step(time, prev), HoldsAt(prev, f), NOT Terminated(time, f) // EC5
14 NEG(HoldsAt(time, f)) :– Step(time, prev), NEG(HoldsAt(prev, f)), NOT Initiated(time, f) // EC6

16 IsAAt(time, x, c) :– HoldsAt(time, IsA(x, c)) // DL1
17 IsAAt(time, x, Neg(c)) :– NEG(HoldsAt(time, IsA(x, c))) // DL2
18 HasAAt(time, x, r, y) :– HoldsAt(time, HasA(x, r, y)) //DL3
```

The above listing leaves off signature declarations, for brevity. The letter f stands for fluents and a for actions. The core relation is HoldsAt(time,f) which can hold true at time because f is Initiated at time (EC3), or was true at the previous time step but not terminated (EC5, frame axiom). Similarly for the negated case.

Notice that ABox assertions are declared as fluents in the first two lines. The last three rules DL1–DL3 provide the glue for translating ABox fluents into timed DL-atoms, which in turn form the ABox for DL-calls, as explained in Section 4.

Notice the difference between Terminated and StronglyTerminated. The former removes HoldsAt(time, f) from the model, the latter inserts **NEG**(HoldsAt(time,f)) into it. That is, this is a three-valued logic. With default negation one can distinguish the three cases.

Fusemate's stratification by time is a good match with the event calculus axioms. The axioms H1 – H3 initiate/terminate fluents with a little delay of 1 time step. This was done so that the Initiates and Terminates predicates can be defined in terms of what HoldsAt at time. The last rule in Example 3 below is an example for that. Without the delay, a non-stratified cycle through default negation in the EC-axioms would result.

Domain dependent axioms. The domain dependent axioms comprise fluents that hold initially and action effect specifications. In the running example, the initial fluents are just the ABox assertions. The box properties are, e.g., HoldsAt(1, IsA(Box(0), FruitBox)) and HoldsAt(1, IsA(Box(5), AndConcept(Box, Exists(Temp, TempClass)))). With the help of the rules DL1–DL3 they become the timed DL-atoms talked about earlier.

Regarding action effect specifications, in it makes sense that loading a box initiates this box to be on the truck, unloading terminates all loaded boxes to be not on the truck, and for all other boxes nothing changes.

Example 3. The following rules specify the effects of the loading and unloading actions. Notice that the OnTruck fluent is not a DL concept (it doesn't have to be).

```
1  // Action declarations
2  case class Load(box: Individual) extends Action
3  case object Unload extends Action
```

```
4 case class SensorEvent(box: Individual, temp: Int) extends Action
5 // Fluent declarations
6 case class OnTruck(box: Individual) extends Fluent
7 // Action/effects
8 Initiates(time, Load(box), OnTruck(box)) :– IsAAt(time, box, Box)
9 StronglyTerminates(time, Unload, OnTruck(box)) :– HoldsAt(time, OnTruck(box))
```
□

Real-world applications require reasoning with concrete domains (numeric types, strings, etc). Extending DLs with concrete domains while preserving satisfiability is possible only under tight expressivity bounds. See [27] for a survey. The proposal here is to use rules and procedural attachments for concrete domains, and the DL reasoner for abstractions.

Example 4. The following rule demonstrates how concrete data can be abstracted into ABox assertions.

```
1 // Classify a box temperature reading as Low or High for this box
2 ( Initiates(time, SensorEvent(box, temp), HasA(box, Temp, High)) AND
3  Terminates(time, SensorEvent(box, temp), HasA(box, Temp, Low)) ) :–
4         Happens(time, SensorEvent(box, temp)), temp > 0
```

In the running example this rule will ascribe at time 20 a Temp attribute to Box_2 for the first time, this way making it from then on a TempBox by the rule in Example 1, and even a KnownTempBox by the rule in Example 2.

The example modelling also contains a symmetric rule with High/Low roles reversed and the condition temp <=0 instead of temp > 0. □

Additional rules. Fusemate provides the user with a number of non-standard operators, see [12]. One of them is the aggregation operator **COLLECT**.

Example 5. Consider the rule

```
1 Unloaded(time+1, boxes.toSet) :–
2         Happens(time, Unload),
3         COLLECT(boxes, box STH HoldsAt(time, OnTruck(box)))
```

This rule aggregates all unloaded boxes into one set, boxes, one tick after Unload time. It is not formulated as a fluent to make it a time*point* property. In the example, the Unload happens at time 50, which leads to Unloaded(51, Set(Box(0), Box(1), Box(2), Box(3), Box(4)). Notice that these are exactly the boxes loaded over time, at timepoints 10, 20, and 30. □

A time*point* property like Unloaded can be used for making determinations about the current state by looking into that state and measuring the time past

Example 6. The following rule automatically flips a box' temperature to High when too much time has passed since unloaded. Notice it is not really an "action" that causes this flip, it is more a matter of circumstances. This is why the rule is formulated in terms of "Initiated/Terminated" (pertaining to fluents), instead of "Initiates/Terminates" (pertaining to actions).

```
Initiated(time+1, HasA(box, Temp, High)) AND Terminated(time+1, HasA(box, Temp, Low)) :–
        TempBox(time, box), // Only boxes with a temperature can rot
        Unloaded(prev < time, boxes) STH (boxes contains box), // See text below
        time — prev >= HighTempGracePeriod,
        NOT HasAAt(time, box, Temp, High) // Loop check
```

The above rule demonstrates Fusemate's comprehension operator. Its application in Unloaded(prev < time, boxes) **STH** (boxes contains box) finds the most recent Unloaded fact(s), at time prev strictly before time and such that boxes contains box. The contains predicate is Scala set library membership. See again [12] for more details. □

Ramification problem. As summarized in [40], the ramification problem is the frame problem for actions with indirect effects, that is to say actions with effects beyond those described explicitly by their associated effect axioms. This frame problem is particularly prominent in the combination with DL, where effects (i.e., fluents) can be entailed implicitly by the DL KB, and possibly in an opaque way. Terminating such a fluent can be futile, it could be re-instated, implicitly or explicitly by materialization.

A good example is the entailment of TempBox(10, Box(0)) and TempBox(10, Box(5)) by the DL KB as discussed after Example 1. Suppose we wish to re-purpose Box(0) and no longer use it for temperature sensitive transport. In terms of the modelling, Box(0) shall no longer belong to the (entailed) concept ∃Temp.TempClass.

The ramification problem has been extensively researched in the EC literature, see again [40]. For instance, one could impose state constraints, if-and-only if conditions, so that terminating an entailed fluent propagates down; or one could use effect constraints that propagate termination of actions to other actions. A first attempt in this direction is a rule that terminates a fluent that entails the property to be removed:

```
Terminated(time+1, HasA(box, Temp, temp)) :–
        RemoveTemp(time, box), // Some condition for removing box Temp
        HasAAt(time, box, Temp, temp), // Attribute to be removed
```

This rule works as expected for Box_2 after explicitly having received a Temp-attribute at time 20, cf. Example 4. It does not work, however, for, e.g., Box_0. As a FruitBox, materialization will immediately restore HasAAt($time + 1$, Box(0), Temp, $temp$).

One way to fix this problem *in the running example* is to terminate *all* concept assertions for the box as any of them might entail a Temp attribute, and only retain that it is a Box:

```
(Terminated(time, IsA(box, concept)) AND Initiated(time, IsA(box, Box))):–
        RemoveTemp(time, box), // Some condition for removing box Temp
        IsAAt(time, box, concept), concept != Box // Concept to be removed
// Similar rule for removing role assertions omitted
```

While this measure achieves the desired effect, it may also remove box properties that could be retained, e.g., the size of the box (if it were part of the running, that is).

The KB revision problem has been studied extensively in database and AI settings. For DLs, there are algorithms for *instance level updates* of an ABox, where, in first-order logic terms, the ABox is a set of ground atoms over known individuals, see [20].

Very recently, Baader etal [7,6] devised algorithms for semantically optimally revising ABoxes that may contain quantifiers (e.g. Box_5 in the running example). All these result are for lightweight description logics, though.

6 Putting it all Together

This section completes the running example with rules for diagnostic reasoning. The rules feature disjunctive heads, classical negation, DL-calls, Scala builtin calls and datatypes. Any Scala datatype qualifies as a Fusemate term, and any Scala case class with a time field of the signature type parameter Time can be an atom.

Example 7. This is a small Scala datatype setup for diagnostic purposes:

```
abstract class Issue
case class TamperedBox(box: Individual) extends Issue // Individual is predefined
case object BrokenCooling extends Issue

abstract class Status extends Atom // Atom is predefined Fusemate atom
case class OK(time: Time) extends Status
case class Anomaly(time: Time, issue: Issue) extends Status
```

The status of the delivery is "ok" or "anomalous". There are two cases of anomalies, (a) the truck cooling is broken or (b) some box has been tampered with. The following rules specify the conditions for that.

```
OK(time) :– Unloaded(time, boxes), NOT Anomaly(time, _)

Anomaly(time, TamperedBox(box)) OR Anomaly(time, BrokenCooling) :–
        Unloaded(time, boxes),
        HasAAt(time, box, Temp, High), boxes contains box

NEG(Anomaly(time, BrokenCooling)) AND NEG(Anomaly(time, TamperedBox(box))) :–
        Unloaded(time, boxes),
        HasAAt(time, box, Temp, Low), boxes contains box

FAIL :–  Anomaly(time, TamperedBox(box)),
        Unloaded(time, unloadedBoxes),
        COLLECT(boxes: List[Individual], x STH (TempBox(time, x), unloadedBoxes contains x)),
        LET(assertions: List[Assertion], boxes map { HasA(_, Temp, High) }),
        DLISSAT(I.aboxAt(time) ++ assertions, tbox)
```

The first rule makes the delivery ok in absence of any anomaly. The second rule observes an anomaly if some unloaded box has a High temperature. The anomaly could be either type, or both, this rule makes a guess. The third and the fourth rule are eliminating guesses. The third rule says that the truck cooling is not broken if evidenced by the existence of a Low temperature box. Moreover, each of these boxes has not been tampered with. The fourth rule is the most interesting one. It eliminates a tampered-box anomaly by considering all unloaded boxes that are known to be equipped with temperature

sensors. The rationale is that if *all* these boxes can consistently be assumed to have High temperature then box tampering is unlikely (broken cooling is more likely).

This reasoning is achieved by collecting in line 13 in the boxes variable the mentioned boxes (TempBox was defined is Example 1). Line 14 assigns to a variable assertions the value of the stated Scala expression for constructing High temperature role assertions for boxes. Finally, the DL-call on line 15 checks the satisfiability of the KB consisting of the current abox temporarily extended with assertions and the static TBox. □

In the running example, the correct diagnosis is Anomaly(51, BrokenCooling). In the course of events, the TempBoxes are Box_0, Box_1, Box_2, and Box_5 (Box_2 becomes one only at time 20.) The unloaded boxes at time 50 are Box_0, Box_1, Box_2, and Box_4. In their intersection, Box_0 and Box_2 have High Temp values, which gives rise to an anomaly. Only the box Box_1 has an unknown Temp value, which is consistent with High and, hence, excludes a TamperedBox anomaly. Moreover, for every box, neither a TamperedBox anomaly nor a negated TamperedBox anomaly is derived.

If the Box_0 sensor reading at time 40 is changed from 10 to -10 then the diagnosis is

```
1 Anomaly(51, TamperedBox(Box(2)))
2 NEG(Anomaly(51, TamperedBox(Box(0))))
3 NEG(Anomaly(51, BrokenCooling))
```

and nothing is known about Box_1.

7 Conclusions

This paper introduced a knowledge representation language that, for the first time, combines the event calculus with description logic in a logic programming framework for model computation. The paper demonstrated the interplay of these three components by means of an elaborated example. The presentation was at a rather concrete level and capitalized on features of the Fusemate system, such as a notion of stratification that is well-suited for timed sequences of events, aggregation constructs and access to host language data structures and functions. Much of the approach, however, should be portable to other logic programming systems that support "time" in a compatible way and with an interface to a DL reasoner, e.g., the DLV system [24].

The modelling in the example emphasised the possibility to distinguish between absent, unknown or known attribute (rule) values, which was enabled by the description logics/rules integration. Being able to distinguish between such existential qualities seems particularly useful in the intended applications where full information is not always given. One might want to go a step further and add "dynamic existentials" to the picture. These are unknown or implicit actions that must have existed to cause observed effects. In a truck transport scenario, for instance, a "traffic jam" action or a "stop at boom gate" action may explain a delay, but only one of them has consequences that are consistent with the actual state. Recovered or speculating such actions can be expressed already with the (implemented) belief revision framework of [11]. Experimenting with that within the framework here is future work.

The perhaps most pressing open issue is the EC ramification problem (Section 5), which is particularly pronounced with the DL integration into the EC. Recent advances on ABox updates might help [7,6].

References

1. Artikis, A., Skarlatidis, A., Portet, F., Paliouras, G.: Logic-based event recognition. Knowl. Eng. Rev. **27**(4), 469–506 (2012). https://doi.org/10.1017/S0269888912000264, https://doi.org/10.1017/S0269888912000264
2. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P. (eds.): Description Logic Handbook. Cambridge University Press (2002)
3. Baader, F., Bauer, A., Baumgartner, P., Cregan, A., Gabaldon, A., Ji, K., Lee, K., Rajaratnam, D., Schwitter, R.: A novel architecture for situation awareness systems. In: Giese, M., Waaler, A. (eds.) Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2009). LNAI, vol. 5607, pp. 77–92. Springer (July 2009). https://doi.org/10.1007/978-3-642-02716-1_7, SAIL-TABLEAUX-09.pdf
4. Baader, F., Borgwardt, S., Lippmann, M.: Temporal conjunctive queries in expressive description logics with transitive roles. In: Pfahringer, B., Renz, J. (eds.) AI 2015: Advances in Artificial Intelligence - 28th Australasian Joint Conference, Canberra, ACT, Australia, November 30 - December 4, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9457, pp. 21–33. Springer (2015). https://doi.org/10.1007/978-3-319-26350-2_3, https://doi.org/10.1007/978-3-319-26350-2_3
5. Baader, F., Ghilardi, S., Lutz, C.: Ltl over description logic axioms. ACM Transactions on Computational Logic - TOCL **13** (01 2008). https://doi.org/10.1145/2287718.2287721
6. Baader, F., Koopmann, P., Kriegel, F., Nuradiansyah, A.: Computing optimal repairs of quantified aboxes w.r.t. static el tboxes. In: CADE-28 - The 28th International Conference on Automated Deduction (2021), https://lat.inf.tu-dresden.de/research/papers/2021/BaKoKrNu-CADE2021.pdf, to appear
7. Baader, F., Kriegel, F., Nuradiansyah, A., Peñaloza, R.: Computing compliant anonymisations of quantified aboxes w.r.t. el policies. In: Pan, J.Z., Tamma, V., d'Amato, C., Janowicz, K., Fu, B., Polleres, A., Seneviratne, O., Kagal, L. (eds.) The Semantic Web – ISWC 2020. pp. 3–20. Springer International Publishing, Cham (2020)
8. Baader, F., Lippmann, M., Liu, H.: Using causal relationships to deal with the ramification problem in action formalisms based on description logics. In: Fermüller, C.G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 82–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
9. Baader, F., Lutz, C., Miličic, M., Sattler, U., Wolter, F.: Integrating description logics and action formalisms: First results. In: Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2. p. 572–577. AAAI'05, AAAI Press (2005)
10. Bagheri Hariri, B., Calvanese, D., De Giacomo, G., Masellis, R., Felli, P., Montali, M.: Description logic knowledge and action bases. Journal of Artificial Intelligence Research **46** (01 2013). https://doi.org/10.1613/jair.3826
11. Baumgartner, P.: Possible Models Computation and Revision – A Practical Approach. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) International Joint Conference on Automated Reasoning. LNAI, vol. 12166, pp. 337–355. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_19, possible-models-IJCAR-2020.pdf
12. Baumgartner, P.: The fusemate logic programming system (system description). In: CADE-28 - The 28th International Conference on Automated Deduction (2021), https://arxiv.org/abs/2103.01395, to appear
13. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper Tableaux. In: Logics in Artificial Intelligence (JELIA '96). No. 1126 in Lecture Notes in Artificial Intelligence, Springer (1996), tableaux-jelia-llncs.pdf
14. Beck, H., Dao-Tran, M., Eiter, T.: LARS: A logic-based framework for analytic reasoning over streams. Artif. Intell. **261**, 16–70 (2018). https://doi.org/10.1016/j.artint.2018.04.003, https://doi.org/10.1016/j.artint.2018.04.003

15. Carral, D., Krötzsch, M.: Rewriting the description logic ALCHIQ to disjunctive existential rules. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. pp. 1777–1783. ijcai.org (2020). https://doi.org/10.24963/ijcai.2020/246, https://doi.org/10.24963/ijcai.2020/246
16. Drescher, C., Thielscher, M.: Integrating action calculi and description logics. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007: Advances in Artificial Intelligence. pp. 68–83. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
17. Eiter, T., Ianni, G., Krennwallner, T., Polleres, A.: Rules and ontologies for the semantic web. In: Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S. (eds.) Reasoning Web: 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures, pp. 1–53. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85658-0_1
18. Eiter, T., Ianni, G., Lukasiewicz, T., Schindlauer, R., Tompits, H.: Combining answer set programming with description logics for the semantic web. Artificial Intelligence **172**(12), 1495–1539 (2008). https://doi.org/10.1016/j.artint.2008.04.002
19. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing pp. 365–385 (1991)
20. Giacomo, G.D., Oriol, X., Rosati, R., Savo, D.F.: Instance-Level Update in DL-Lite Ontologies through First-Order Rewriting. J. Artif. Intell. Res. **70**, 1335–1371 (2021). https://doi.org/10.1613/jair.1.12414
21. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: combining logic programs with description logic. In: Hencsey, G., White, B., Chen, Y.R., Kovács, L., Lawrence, S. (eds.) Proceedings of the Twelfth International World Wide Web Conference, WWW 2003, Budapest, Hungary, May 20-24, 2003. pp. 48–57. ACM (2003). https://doi.org/10.1145/775152.775160, https://doi.org/10.1145/775152.775160
22. Kowalski, R.A., Sergot, M.J.: A Logic-based Calculus of Events. New Generation Computing **4**(1), 67–95 (1986). https://doi.org/10.1007/BF03037383
23. Lee, J., Palla, R.: Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. Journal of Artificial Intelligence Research **43**, 571–620 (Jan 2012). https://doi.org/10.1613/jair.3489
24. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3), 499–562 (Jul 2006). https://doi.org/10.1145/1149114.1149117, https://doi.org/10.1145/1149114.1149117
25. Lin, F.: Situation calculus. In: van Harmelen, F., Lifschitz, V., Porter, B.W. (eds.) Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 649–669. Elsevier (2008). https://doi.org/10.1016/S1574-6526(07)03016-7, https://doi.org/10.1016/S1574-6526(07)03016-7
26. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in prolog: The dlog system. Theory Pract. Log. Program. **9**(3), 343–414 (2009). https://doi.org/10.1017/S1471068409003792, https://doi.org/10.1017/S1471068409003792
27. Lutz, C.: Description Logics with Concrete Domains - A Survey. In: Balbiani, P., Suzuki, N., Wolter, F., Zakharyaschev, M. (eds.) Advances in Modal Logic 4, papers from the fourth conference on "Advances in Modal logic," held in Toulouse, France, 30 September - 2 October 2002. pp. 265–296. King's College Publications (2002)
28. Manthey, R., Bry, F.: SATCHMO: a theorem prover implemented in Prolog. In: Lusk, E., Overbeek, R. (eds.) Proceedings of the 9^{th} Conference on Automated Deduction, Argonne, Illinois, May 1988. Lecture Notes in Computer Science, vol. 310, pp. 415–434. Springer (1988)

29. Miller, R., Shanahan, M.: Some alternative formulations of the event calculus. In: Kakas, A.C., Sadri, F. (eds.) Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part II, pp. 452–490. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45632-5_17
30. Motik, B., Sattler, U., Studer, R.: Query answering for owl-dl with rules. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) The Semantic Web – ISWC 2004. pp. 549–563. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
31. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. Artif. Intell. Res. **36**, 165–228 (2009). https://doi.org/10.1613/jair.2811, https://doi.org/10.1613/jair.2811
32. Mueller, E.T.: Event calculus reasoning through satisfiability. Journal of Logic and Computation **14**(5), 703–730 (2004)
33. Özçep, Ö.L., Möller, R., Neuenstadt, C.: A stream-temporal query language for ontology based data access. In: Lutz, C., Thielscher, M. (eds.) KI 2014: Advances in Artificial Intelligence. pp. 183–194. Springer International Publishing, Cham (2014)
34. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Spaccapietra, S. (ed.) Journal on Data Semantics X. pp. 133–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
35. Przymusinski, T.C.: Chapter 5 - on the declarative semantics of deductive databases and logic programs. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 193 – 216. Morgan Kaufmann (1988). https://doi.org/https://doi.org/10.1016/B978-0-934613-40-8.50009-9, http://www.sciencedirect.com/science/article/pii/B9780934613408500099
36. Rosati, R.: Dl+log: Tight integration of description logics and disjunctive datalog. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006. pp. 68–78. AAAI Press (2006), http://www.aaai.org/Library/KR/2006/kr06-010.php
37. Sakama, C.: Possible Model Semantics for Disjunctive Databases. In: Kim, W., Nicholas, J.M., Nishio, S. (eds.) Proceedings First International Conference on Deductive and Object-Oriented Databases (DOOD-89). pp. 337–351. Elsevier Science Publishers B.V. (North–Holland) Amsterdam (1990)
38. Sakama, C., Inoue, K.: An Alternative Approach to the Semantics of Disjunctive Logic Programs and Deductive Databases. Journal of Automated Reasoning **13**, 145–172 (1994)
39. The Scala Programming Language, https://www.scala-lang.org
40. Shanahan, M.: The event calculus explained. In: Wooldridge, M.J., Veloso, M. (eds.) Artificial Intelligence Today: Recent Trends and Developments, pp. 409–430. Springer Berlin Heidelberg, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48317-9_17
41. Tsilionis, E., Artikis, A., Paliouras, G.: Incremental event calculus for run-time reasoning. In: Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems. p. 79–90. DEBS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3328905.3329504