Possible Models Computation and Revision – A Practical Approach

Peter Baumgartner[1,2](✉)

[1] Data61/CSIRO, Canberra, Australia
Peter.Baumgartner@data61.csiro.au
[2] The Australian National University, Canberra, Australia

Abstract. This paper describes a method of computing plausible states of a system as a logical model. The problem of analyzing state-based systems as they evolve over time has been studied widely in the automated reasoning community (and others). This paper proposes a specific approach, one that is tailored to situational awareness applications. The main contribution is a calculus for a novel specification language that is built around disjunctive logic programming under a possible models semantics, stratification in terms of event times, default negation, and a model revision operator for dealing with incomplete or erroneous events – a typical problem in realistic applications. The paper proves the calculus correct wrt. a formal semantics of the specification language and it describes the calculus' implementation via embedding in Scala. This enables immediate access to rich data structures and external systems, which is important in practice.

1 Introduction

This paper is concerned with logic-based modeling and automated reasoning for estimating the current state of a system as it evolves over time. The main motivation is situational awareness [12], which requires the ability to understand and explain a system's state, at any time, and at a level that matters to the user, even if only partial or incorrect information about the external events leading to that state is available. In a supply chain context, for example, one cannot expect that events are reported correctly and in a timely manner. Sensors may fail, transmission channels are laggy, reports exist only in paper form, not every player is willing to share information, etc. Because of that, it is often impossible to know with full certainty the actual state of the system. The paper addresses this problem and proposes to instead derive a set of *plausible candidate states* as an approximation of ground truth. The states may include consequences relevant for situational awareness, e.g., that a shipment will be late. A human operator may then make decisions or provide additional details, this way closing the loop.

The plausible candidate states are represented as models of a logical specification and a given a set of external timestamped events. The proposed modeling

This research is supported by the Science and Industry Endowment Fund.

paradigm is logic programming, and models are computed in a bottom-up way. It adopts notions of stratification, default negation and a possible model semantics for its (disjunctive) program rules. Stratification is in terms of event time, with increasing time horizons for anytime reasoning; default negation is needed to reason in absence of information such as event reports; disjunctions are needed to derive alternate candidate states. In order to deal with less-than-perfect event data, the modeling language features a novel model revision operator that allows the programmer to formulate conditions under which a model computation with a corrected set of events should be attempted in an otherwise inconsistent state. The following informal overview illustrates these features.

1.1 Main Ideas and Design Rationale

A model, or program, is comprised of a set of rules of the form *head* $\leftarrow$ *body*. The *head* can be a non-empty disjunction of atoms, or a **fail** head. The former rules open the solution (models) space for a fixed set of external events, while **fail** head rules limit it and specify amended event sets for new solution attempts. The *body* is a conjunction of atoms and negated (via "**not**") conjunctions of atoms. Negation is "default negation", i.e., a closed world assumption is in place for evaluating the latter. Rules may contain first-order variables and must be range restricted. This guarantees that only ground heads can be derived from ground facts when a rule is evaluated in a bottom-up way. Our notion of range restriction is somewhat non-standard, though, and permits extra variables inside negation. These variables are implicitly existentially quantified ("**not** $\exists x \ldots$ ").

We need, however, syntactic restrictions that enforce stratification in terms of "time". This entails that rule evaluation does not depend on facts from the future. In fact, this is a reasonable assumption for situational awareness, whose any-time character requires to understand the current situation based on the available information up to now *only.* Technically, every atom must have a dedicated "time" argument, a $\mathbb{N}$-sorted variable, which, together, with earlier-than constraints (via "$<$" or "$\leq$") enforces stratification. The details of that will have to wait until later (Definition 1). An example rule is

$$\mathsf{hungry}(t,x) \vee \mathsf{thirsty}(t,x) \leftarrow \mathsf{get\_up}(t,x), \mathbf{not}(t-6 \leq s, s \leq t, \mathsf{meal}(s,x)) \ . \quad (1)$$

It could say "if x gets up at time t and didn't have a meal in 6 hours prior then x is hungry or thirsty at t, *or both*". A set of facts, say, $\{\mathsf{get\_up}(8,\mathsf{bob}), \mathsf{meal}(12,\mathsf{bob})\}$ then entails $\mathsf{hungry}(8,\mathsf{bob}) \vee \mathsf{thirsty}(8,\mathsf{bob})$. Notice that in the relevant rule instance the negated body element $\mathbf{not}(8-6 \leq s, s < 8, \mathsf{meal}(s,\mathsf{bob}))$ is satisfied by the facts (using the closed-world assumption), as for the only relevant meal-instance $\mathsf{meal}(12,\mathsf{bob})$ the arithmetic constraint is false. The possible model semantics [26], which we adopt, interprets disjunctions (also) inclusively. Each resulting case $\{\mathsf{hungry}(8,\mathsf{bob})\}$, $\{\mathsf{thirsty}(8,\mathsf{bob})\}$ and $\{\mathsf{hungry}(8,\mathsf{bob}), \mathsf{thirsty}(8,\mathsf{bob})\}$ together with the facts yields a possible model.

Stratification in terms of time makes default negation with existentially quantified variables possible; no need to look into the future. We impose a secondary kind of stratification that also makes it more *efficient*. It rests on distinguishing two types of atoms: EDB atoms and IDB atoms (extensional/intensional database, respectively). EDB atoms are for external events, the given facts, and IDB atoms are for derived facts. A disjunctive *head* can contain IDB atoms only. Now, an IDB atom within a negation has to be strictly earlier ($<$) than the head, while an EDB atom within a negation can be non-strictly earlier ($\leq$). This make sure that a truth value for a negated expression cannot change later, in the course of rule evaluation in increasing time order. The rule (1) above is stratified if meal is EDB, otherwise "$s \leq t$" would have to be "$s < t$". A rule like $\mathsf{hungry}(t,x) \leftarrow \mathsf{get\_up}(t,x), \textbf{not}\ \mathsf{hungry}(t,x)$ cannot be stratified.

Rules with **fail** heads enable the programmer to specify when a (partial) model candidate is unsatisfiable *and to say how to potentially fix this situation.* A rule of the form $\textbf{fail}() \leftarrow \mathit{body}$ without arguments to **fail** is a usual integrity constraint, e.g.:

$$\textbf{fail}() \leftarrow \mathsf{hungry}(t,x), \mathsf{eat}(s,x), t-4 \leq s, s < t \tag{2}$$

The rule (2) rejects that x is both hungry and has eaten within the last 4 hours. Together with rule (1) and the EDB $\{\mathsf{eat}(7,\mathsf{bob}), \mathsf{get\_up}(8,\mathsf{bob})\}$ one obtains the sole possible model $\{\mathsf{eat}(7,\mathsf{bob}), \mathsf{get\_up}(8,\mathsf{bob}), \mathsf{thirsty}(8,\mathsf{bob})\}$.

The second usage is of the form $\textbf{fail}(+a, -b) \leftarrow \mathit{body}$, where a and b are EDB atoms with timestamps not in the future. When the rule body is satisfied, the model computation restarts with a added and b removed from the EDB. For example, the rule

$$\textbf{fail}(-\mathsf{eat}(s,x)) \leftarrow \mathsf{get\_up}(t,x), \mathsf{eat}(s,x), t-1 \leq s, s < t \tag{3}$$

rejects a model candidate where x has eaten within one hour before getting up. The rule correspondingly removes the eat event from the EDB and the model computation restarts. One could further add

$$\textbf{fail}(-\mathsf{get\_up}(t,x)) \leftarrow \mathsf{get\_up}(t,x), \mathsf{eat}(s,x), t-1 \leq s, s < t \tag{4}$$

to (1)–(3) as an alternative to fix the problem. The principle is that as soon as the earliest **fail** head is derived, the model candidate at that time is given up. Then alternate model computations are started for *all* **fail** heads derivable for that time. Later times are not considered. In the example, thus, both restarts prescribed by rules (3) and (4) are tried.

1.2 Related Work and Novelty

Assigning models to logic programs as their intended meaning has been studied for decades. We only mention the stable models semantics [14,18], its extension for disjunctive programs [11,15], and the possible model semantics [26,27] as the most relevant in the following discussion. Both ascribe meaning to a given program in terms of minimal models, but differ in the way disjunctive rule heads are interpreted (exclusive vs. inclusive, respectively).

Most reasoning tasks around stable models are rather complex, e.g., model existence for propositional disjunctive programs is Σ_2^P-complete [10]. This complexity translates into generate-and-test algorithms even without default negation. For instance, the tableau calculus in [22] for negation-free programs generates in its branches model candidates, whose minimality need to be tested by a subsequent theorem prover call. Another need for generate-and-test algorithms for stable models comes from default negation. In the general case, these algorithms need to guess a stable model candidate and verify its minimality for a certain negation-free program obtained by simplification with this candidate, the Gelfond-Lifschitz transformation.

In contrast, our approach avoids the intricacies of generate-and-test algorithms. This is achieved by using the possible model semantics [26] and a specific concept of stratification for dealing with default negation. The latter fits in the framework of local stratification [25]. A similar concept of stratification by time has been employed for expressing greedy algorithms in Datalog [30]). The usual stratified case, by predicates, but without quantification within negation was already been considered in [26].

As indicated in Sect. 1.1 our language features a **fail** operator for model revision. This feature is the one that possible stands out most among the other mentioned. A rule $\textsf{\textbf{fail}}(-\mathsf{p}(x)) \leftarrow \mathsf{q}(x), \mathsf{p}(x)$ applied to the facts $\{\mathsf{q}(\mathsf{a}), \mathsf{p}(\mathsf{a})\}$ derives the model $\{\mathsf{q}(\mathsf{a})\}$. This cannot be achieved without belief revision, as given facts have to be satisfied.

Belief revision [1,24] is the process of changing beliefs to take into account a new piece of information. It has also been studied extensively in a logic programming context and in a general way. For instance, Schwind and Inoue [29] consider the problem of revision by a *program* in a rather expressive setting, generalized logic programs equipped with stable model semantics. The perhaps closest approach to ours are the revision specifications of [19,20]. Revision programs generalize logic programming with stable model semantics by an explicit deletion operator. Each revised model is obtained from the initial interpretation by means of insertions and deletions specified by a Gelfond-Lifschitz type reduced program. In that way, our approach is related, but simpler, as it revises only the EDB and does not require a generate-and-test algorithm. On the other hand, the semantics of our revisions takes timestamps into account, so that intended revisions are only those that are derivable "now".

The focus on the paper is not on situational awareness as such. We merely mention that the problem has attracted interest from a logical perspective. In earlier work [2] we proposed bottom-up model computation with a Hyper Tableaux prover [4,23] as a component for data aggregation. In a related context of conformance checking, the authors of [7] propose if-then rules for validating process execution traces by means of a Prolog interpreter. Other approaches for conformance checking include planning [9] and diagnosis of discrete dynamical systems [8,21].

To sum up, the main novelty of our approach lies in the combination of the possible model semantics with specific concepts of stratification and model revision. The combination is designed to enable simple fixpoint algorithms that are sound and complete for a not too complicated declarative semantics. This is the main theoretical contribution.

On the practical side we offer a (publicly available) implementation of our calculus, as a shallow embedding into the Scala programming language. Somewhat related, a shallow embedding into Scala has been used for monitoring event streams over Allen's temporal interval logic [17]. Yet, it is an uncommon implementation technique for automated reasoning systems. The practical advantages are described in Sect. 6.

2 Preliminaries

We assume the reader is familiar with basic notions of first-order logic and answer set programming. See [6] and [13], respectively, for introductory texts.

A first-order logic signature $\Sigma = \Sigma_P \uplus \Sigma_F$ is comprised of predicate symbols Σ_P and function symbols Σ_F of fixed arities. We assume $\mathbb{N} \subseteq \Sigma_F$, i.e., the natural numbers are also constants of the logical language, and that Σ_P contains the *arithmetic predicate symbols* $\Sigma_{\mathbb{N}} = \{<, \leq, =, \neq\}$. The *ordinary predicate symbols* are $\Sigma_P \setminus \Sigma_{\mathbb{N}}$. Let $\mathcal{X}$ be a countably infinite set of variables. Instead of introducing a two-sorted signature we assume informally that all terms and formulas over Σ and $\mathcal{X}$ are built in a sorted way.

The letters s and t usually stand for terms, x and y stand for variables, and p and q for ordinary predicate symbols. We speak of *ordinary atoms* and *arithmetic atoms* depending on whether the predicate symbol is ordinary or arithmetic, respectively. For a set A of atoms let $\mathit{ord}(A)$ be the set of all ordinary atoms in A.

Intuitively, $\mathbb{N}$ represents timestamps (points in time), and $<$ and $\leq$ stand for the strict and non-strict earlier-than relationships, respectively. We assume every ordinary predicate symbol has arity ≥ 1 and its, say, first argument ranges over $\mathbb{N}$. For any ordinary atom $a = p(t_1, \ldots, t_n)$ let $\mathit{time}(a) = t_1$ be its timestamp. The function symbols Σ_F may contain arithmetic operations as needed to compute with timestamps, but Σ_F may not contain uninterpreted operators with $\mathbb{N}$ as the result sort.

We assume the ordinary predicate symbols are partitioned as $\Sigma_P \setminus \Sigma_{\mathbb{N}} = \Sigma_{\mathrm{EDB}} \uplus \Sigma_{\mathrm{IDB}}$. The symbols in Σ_{EDB} are called *extensional database (EDB) predicates*, and the symbols in Σ_{IDB} are the *intensional database (IDB) predicates*. An *EDB* is a finite set of ground Σ_{EDB}-atoms, and an *IDB* is a finite set of ground Σ_{IDB}-atoms. We may think of an EDB as a timestamped sequence of external events, and an IDB as higher-level conclusions derived from that EDB. Below we will exploit this distinction for computing models in a stratified way and for defining a model revision operator.

As usual, a substitution σ is a mapping from the variables to terms. A substitution is identified with its homomorphic extension to terms. Substitution application is written postfix, i.e., we write $t\sigma$ instead of $\sigma(t)$. The *domain of* σ is the set $\mathit{dom}(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$ and is always assumed to be finite.

When z is a term, an atom, a sequence, or a set of those, let $var(z)$ denote the set of variables occurring in z. We say that z is *ground* if $var(z) = \emptyset$. A substitution γ is a *grounding substitution for z* iff $z\gamma$ is ground. In this case $z\gamma$ is also called a *ground instance of z (via γ)*. Let $gnd(z)$ denote the set of all ground instances of z.

3 Stratified Programs

We are now in a position to define our main modeling tool, a variation on if-then rules as popularized in the area of disjunctive logic programming.

A *positive body* is a list $\vec{b} = b_1, \ldots, b_k$ of atoms with $k \geq 0$. If $k = 0$ then $\vec{b}$ is *empty* otherwise it is *non-empty*. (The list represents a conjunction.) A *negative body literal* is an expression of the form $\textsf{not}\,\vec{b}$, where $\vec{b}$ is a non-empty positive body. A *body* is a list $b_1, \ldots, b_k, \textsf{not}\,\vec{b}_{k+1}, \ldots, \textsf{not}\,\vec{b}_n$ comprised of a (possibly empty) positive body and (possibly zero) negative body literals. It is *variable free* if $var(b_1, \ldots, b_k) = \emptyset$. A *head* is one of the following:

(a) an *ordinary head*: a disjunction $h_1 \vee \cdots \vee h_m$ of IDB atoms, for some $m \geq 1$, or

(b) a *fail head*: an expression of the form $\textsf{fail}(\vec{e})$ where $\vec{e} = \pm_1 e_1, \ldots, \pm_k e_k$, for some $k \geq 0$, EDB atoms e_i and $\pm_i \in \{+, -\}$. If $k = 0$ then $\vec{e}$ is the empty sequence ε, and $\textsf{fail}(\epsilon)$ is usually written as $\textsf{fail}()$.

A *rule* consist of a head H and a body and is commonly written as an implication

$$H \leftarrow b_1, \ldots, b_k, \textsf{not}\,\vec{b}_{k+1}, \ldots, \textsf{not}\,\vec{b}_n \ . \tag{5}$$

By an *ordinary rule* (*fail rule*) we mean a rule with an ordinary head (fail head), respectively. A *fail set* is a (possibly empty) set of ground fail heads.

Let r be a rule (5) and $\vec{b} = b_1, \ldots, b_k$ its positive body. We say that r is *variable free* iff $var(H) \cup var(\vec{b}) = \emptyset$. This notion of variable-freeness is justified by the fact that the extra variables $var(\vec{b}_i) \setminus var(\vec{b})$ in the negative body literals $\textsf{not}\,\vec{b}_i$ are implicitly existentially quantified, see Definition 5 below. We say that *r' is a variable free instance of r via σ* iff $r' = r\sigma$ is variable free and $dom(\sigma) = var(H) \cup var(\vec{b})$. Notice that σ must not act on the extra variables as these are shielded by quantification.

A *program* is a set of rules. It is *variable free* if all of its rules are. Semantically, every program R stands for the (possibly infinite) variable free program $\mathit{vfinst}(R)$ that is obtained by taking all variable free instances of all rules in R.

The rules are to be evaluated in a bottom-up way. If a current model candidate satisfies a rule body then its head needs evaluation. An ordinary rule extends the current model according to the possible model semantics as explained below and a fail rule rejects the current model. If a fail rule's head is $\textsf{fail}()$ it acts like a traditional rule with an empty head, as an integrity constraint. If the argument list $\vec{e}$ is non-empty the fail rule "fixes" the current EDB by adding ("+") or removing ("−") EDB atoms and starting a new model computation.

In order to admit effective model computation our rules will be stratified. Stratification means range-restrictedness and other restrictions on variables and negation.

Definition 1 (Stratified rule). *Let r be a rule (5) with positive body $\vec{b} = b_1, \ldots, b_k$ and y be a variable. The rule r is* stratified wrt. y *if there is a $b \in \vec{b}$ such that $time(b) = y$ and the following holds:*

(i) $var(\vec{b}) \subseteq var(ord(\vec{b}))$,
(ii) *for every ordinary atom $b \in \vec{b}$,*
$time(b) = y$ or $time(b) = x$ and $x \lhd y \in \vec{b}$ for some variable x and $\lhd \in \{<, \leq\}$,
(iii) *every negative body literal* $\textsf{\textbf{not}}\,\vec{b}_{k+1}, \ldots, \textsf{\textbf{not}}\,\vec{b}_n$ *is stratified, and*
(iv) *the head H is stratified.*

In the above, a negative body literal $\textsf{\textbf{not}}\,\vec{b}_i$ *is stratified if the following holds:*

(i) $var(\vec{b}_i) \setminus var(\vec{b}) \subseteq var(ord(\vec{b}_i))$,
(ii) *for every EDB atom $b \in \vec{b}_i$,*
$time(b) = y$ or $time(b) = x$ and $x \lhd y \in \vec{b}_i$ for some variable x and $\lhd \in \{<, \leq\}$, and
(iii) *for every IDB atom $b \in \vec{b}_i$,*
$time(b) = x$ and $x < y \in \vec{b}_i$ for some variable x.

The head H is stratified if the following holds:

(i) $var(H) \subseteq var(ord(\vec{b}))$,
(ii) *if H is an ordinary head $h_1 \vee \cdots \vee h_m$ then $time(h_1) = \cdots = time(h_m) = y$.*
(iii) *if H is a fail head* $\textsf{\textbf{fail}}(\vec{e})$ *then for all $\pm e \in \vec{e}$, $time(e)$ is an arithmetic expression and $time(e) \lhd y \in \vec{b}$, for some $\lhd \in \{<, \leq\}$.*

A rule is stratified *if it is stratified wrt. some variable y. A program is stratified if each of its rules is stratified.*[1]

Definition 1 expresses conditions on rules in terms of *time-restrictedness* and *range-restrictedness*. The variable y stands for the latest of all timestamps among all timestamps of the ordinary atoms in the rule body. This is made sure by constraints $x \lhd y$ in the various parts of the definition where $\lhd \in \{<, \leq\}$. More precisely, ordinary atoms in the positive body are timestamped "$\leq$"; ordinary heads are timestamped "y" so that no literals timestamped in the past can be inserted into the model (this would defy stratification); and restarts can modify only the past. For the ordinary atoms in negative body literals we distinguish between EDB and IDB atoms. EDB atoms cannot be derived in heads of rules, which affords "$\leq$", whereas IDB atoms must be "$<$".

[1] Usually, stratification is defined as a property of the program as a whole, via its call-graph.

```
(1) In(time, obj, cont) :-
        Load(time, obj, cont)

(2) In(time, obj, cont) :-
        In(time, obj, c),
        In(time, c, cont)

(3) In(next, obj, cont) :-
        In(time, obj, cont),
        Step(next, time),
        not(Unload(next, obj, cont)),
        not(In(time, obj, c),
             Unload(next, c, cont))

(4) fail(+Unload((time + prev)/2,
                 cont, c)) :-
        Unload(time, obj, cont),
        In(time, cont, c),
        Load(t, cont, c), t < time,
        Step(time, prev)

(5) fail() :-
        Unload(time, obj, cont),
        Step(time, prev),
        not(In(prev, obj, cont))

(6) fail() :-
        Unload(time, obj, cont),
        not(Load(t, obj, cont), t < time)

(7) fail(−Unload(time, obj, cont),
         +Unload(time, o, cont)) :-
        Unload(time, obj, cont),
        not(Load(t, obj, cont), t < time),
        t < time,
        Load(t, o, cont),
        SameBatch(t, b),
        if((b contains obj) and (b contains o))

(8) fail(+Load(t, obj, cont)) :-
        Unload(time, obj, cont),
        not(Load(t, obj, cont), t < time),
        t < time,
        Load(t, o, cont),
        SameBatch(t, b),
        if((b contains obj) and (b contains o))

(9) fail(−Load(t, o, cont),
         +Load(t, obj, cont)) :-
        not(Load(t, obj, cont), t < time),
        Load(t, o, cont),
        t < time,
        SameBatch(t, b),
        if((b contains obj) and (b contains o))
```

Fig. 1. Supply chain program. See Example 2 for explanations.

The remaining conditions force range-restrictedness. In the first part, condition (i) says that every variable in a positive body atom appears also in some ordinary positive atom; similarly for condition (i) for heads. Condition (i) for negative body atoms says that every extra variable in a negative body atom appears also in some of its ordinary body atoms. Together these conditions make sure that matching a rule's ordinary atoms against a ground candidate model always removes all variables. This way, all arithmetic expressions can be evaluated and only ground heads can be derived.

Example 1 (Stratified rule). Let $\Sigma_{\text{EDB}} = \{\mathsf{p}\}$ and $\Sigma_{\text{IDB}} = \{\mathsf{d}\}$. The rule $\mathsf{d}(x_3, x_1) \leftarrow x_1 < x_3, \mathsf{p}(x_1), \mathsf{p}(x_3), \textbf{not}\,(x_1 < x_2, x_2 < x_3, \mathsf{p}(x_2))$ is stratified wrt. x_3 $(= \mathit{time}(\mathsf{p}(x_3)))$. It collects in d the timestamps between *consecutive* p-events. For example, given the set $I = \{\mathsf{p}(2), \mathsf{p}(4), \mathsf{p}(7), \mathsf{p}(13)\}$, the rule when applied exhaustively derives $\mathsf{d}(4, 2)$, $\mathsf{d}(7, 4)$ and $\mathsf{d}(13, 7)$ but not, e.g, $\mathsf{d}(7, 2)$. The extra variable x_2 in its negative body literal is implicitly existentially quantified. □

Example 2 (Supply chain). The program in Fig. 1 illustrates a possible use of our approach in a supply chain application. It is written in the concrete input syntax of our implementation, the Fusemate system (Sect. 6).

The signature is $\Sigma_{\text{EDB}} = \{\mathsf{Load}, \mathsf{Unload}, \mathsf{SameBatch}\}$ and $\Sigma_{\text{IDB}} = \{\mathsf{In}\}$. A fluent $\mathsf{Load}(\mathsf{time}, \mathsf{obj}, \mathsf{cont})$ expresses that at the given time an object obj is loaded into a container cont, similarly for Unload. A fluent $\mathsf{In}(\mathsf{time}, \mathsf{obj}, \mathsf{cont})$ says that at the given time an object obj is inside the container cont.

With this interpretation, the rules (1) and (2) for the In relation should be obvious. Rule (3) is a frame axiom for the In relation. That is, it states when an In-fluent carries over to the next timestamp: an object remains in a container if neither it nor a container containing it is unloaded from the container. The body atom $\mathsf{Step}(\mathsf{time}, \mathsf{prev})$ holds true if prev is the most recent timestamp preceding time. The Step relation is "built-in" into Fusemate for convenience.

Rule (4) fixes the problem of a "missing" unloading event by inserting one into the EDB at a speculated time $(\mathsf{time} + \mathsf{prev})/2$. This rule will become clearer in Example 3 below, where we discuss the program in conjunction with a concrete EDB.

Rule (5) says that only items that are in a container can be unloaded in the next step. Rule (6) demands loading prior to unloading. The other rules will also be discussed below. □

4 Semantics

The possible model semantics [26,27] associates to a disjunctive program sets of possible facts that might have been true in the actual world. (This is already a good fit for situational awareness.) We extend it to our stratified programs with fail rules.

Let $\text{Th}(\Sigma_{\mathbb{N}})$ be the set of all ground time atoms that are true in the standard model of natural number arithmetic. For a set A of ground ordinary atoms define $\mathcal{I}(A) = A \cup \text{Th}(\Sigma_{\mathbb{N}})$ which represents the Herbrand Σ-interpretation that assigns true to a ground atom a if and only if $a \in \mathcal{I}(A)$.

Definition 2 (Rule semantics). *A set A of ground ordinary atoms* satisfies *a variable free body $B = b_1, \ldots, b_k, \textbf{not}\, \vec{b}_{k+1}, \ldots, \textbf{not}\, \vec{b}_n$, written as $A \models B$, if $\{b_1, \ldots, b_k\} \subseteq \mathcal{I}(A)$ and $(gnd(\vec{b}_{k+1}) \cup \cdots \cup gnd(\vec{b}_n)) \cap \mathcal{I}(A) = \emptyset$.*

The set A satisfies *a variable free ordinary rule $h_1 \vee \cdots \vee h_m \leftarrow B$ if $A \models B$ entails $\{h_1, \ldots, h_m\} \cap \mathcal{I}(A) \neq \emptyset$. It is a* model *of a set R of variable free ordinary rules, written as $A \models R$, iff it satisfies every rule in R.*

The fail set *of A and a set of variable free fail rules R is the set $F = \{\textbf{fail}(\vec{e}) \mid$ there is a rule $\textbf{fail}(\vec{e}) \leftarrow B \in R$ such that $A \models B\}$. This is written as $A \models_R^{\textbf{fail}} F$.*

Satisfaction of a variable free body according to Definition 2 is equivalent to satisfaction of the first-order logic formula $b_1 \wedge \cdots \wedge b_k \wedge \neg(\exists\, \vec{x}_{k+1}.\ \wedge \vec{b}_{k+1}) \wedge \cdots \wedge \neg(\exists\, \vec{x}_n.\ \wedge \vec{b}_n)$ in the interpretation $\mathcal{I}(A)$, where $\vec{x}_j = var(\vec{b}_j)$, for all $j = k+1, \ldots, n.$.

Definition 2 is, in fact, somewhat more general than needed for defining the possible models semantics of logic programs. Possible models interpret disjunctive heads inclusively, in all possible ways. This is expressed in the following definition.

Definition 3 (Split program [26,27]). *Let R be a variable free program. A* split program of R *is any program obtained from R by replacing every ordinary rule $h_1 \vee \cdots \vee h_m \leftarrow B$ by the normal ordinary rules (called split rules) $h \leftarrow B$, for every $h \in \mathcal{H}$, where $\mathcal{H}$ is some non-empty subset of $\{h_1, \ldots, h_m\}$.*

In [26,27], Sakama et al. define the possible model semantics (also) for disjunctive programs without negation. Our stratified case admits a similar definition.

For any program R let R^+ (R^-) be the set of all ordinary (fail) rules of R, respectively.

Definition 4 (Satisfication of stratified programs). *Let P be a stratified program, F a fail set, E an EBD and I an IDB. We write $(E, I) \models_P^{\mathsf{fail}} F$ if there is a split program S of vfinst(P) such that all of the following hold:*

(i) $E \cup I \models S^+$ *($E \cup I$ is a model of the ordinary rules)*
(ii) $E \cup J \models S^+$ *for no* $J \subsetneq I$ *(I is minimal)*
(iii) $E \cup I \models_{S^-}^{\mathsf{fail}} F$ *(F is the triggered fail heads)*

If $F = \emptyset$ then we say that (E, I) satifies P *and writte $(E, I) \models P$.*

As an example (without time), if $E = \{p\}$ and $R = \{q \vee r \leftarrow p,\ q \leftarrow r,\ s \leftarrow s\}$ then $\{p, q, s\} \models R$, but only $(\{p\}, \{q\})$ and $(\{p\}, \{r, q\})$ satisfy R in the sense of Definition 4.

The purpose of a program P is to compute all extensions (E, I) of a given EDB E that satisfy P. For failed such attempts, P also specifies ways to revise E, if any, as early as possible, leading to new tries. The following Definition 5 make this precise.

For a ground fail head $\mathsf{fail}(\vec{e})$ and ground EDB E let $upd(E, \vec{e})$ be the EDB obtained from E by first adding all EDB atoms e such that $\vec{e}$ contains the expression $+e$, and then deleting all all EDB atoms e such that $\vec{e}$ contains $-e$. For any set A of ground ordinary atoms and $t \in \mathbb{N}$ let $A_{\leq t} = \{a \in A \mid time(a) \leq t\}$; analogously for $A_{<t}$.

Definition 5 (Possible models of stratified programs). *Let P be a stratified program and E_{init} an EDB. Let $\mathcal{E}$ be the smallest set of EDBs containing E_{init} and satisfying, for all $E \in \mathcal{E}$, timestamps t in E, IDBs I and fail sets F:*

If $(E_{\leq t}, I) \models_P^{\mathsf{fail}} F$ and there is no $J \subseteq I$ and $G \neq \emptyset$ such that $(E_{<t}, J) \models_P^{\mathsf{fail}} G$ then $\{upd(E, \vec{e}) \mid \mathsf{fail}(\vec{e}) \in F \text{ and } \vec{e} \neq \varepsilon\} \subseteq \mathcal{E}$.

The set $\mathcal{E}$ is called the restart EDBs induced by P and E_{init}. *Any pair (E, I) such that $E \in \mathcal{E}$ and $(E, I) \models P$ is called a* possible model of P and E_{init}, *written as $(E_{\mathsf{init}}, E, I) \models P$. Let $mods_P(E_{\mathsf{init}}) = \{(E, I) \mid (E_{\mathsf{init}}, E, I) \models P\}$ be all such possible models.*

The set $\mathcal{E}$ in Definition 5 contains a restart EDB E apart from E_{init} if and only if E is obtained from some *earliest time* fail set F from another restart EDB in $\mathcal{E}$. This excludes fail sets that may otherwise be additionally derivable at a later time. This was a design decision in support of "anytime" reasoning, for not having to consider future events.

Example 3 (Supply chain, Example 2 continued). Consider the following EDB E_{init} consisting of loading and unloading events:

$\mathsf{SameBatch}(10, \mathsf{Set}(\mathsf{tomatoes}, \mathsf{apples}))$	$\mathsf{Load}(40, \mathsf{container}, \mathsf{ship})$
$\mathsf{Load}(10, \mathsf{tomatoes}, \mathsf{pallet})$	$\mathsf{Unload}(60, \mathsf{apples}, \mathsf{pallet})$
$\mathsf{Load}(20, \mathsf{pallet}, \mathsf{container})$	

The intuitive meaning of the Load atoms between times 10 and 40 should be obvious. All what is reported at time 60 is that apples are unloaded from the pallet. However, this is suspicious from a (practical) completeness and consistency perspective. First, it can be alleged that some unloading events went under unreported. Before an item (apples) can be unloaded from a pallet that was loaded earlier into a container, the pallet needs to be unloaded from the container first, and that container must have been unloaded from the ship. Such reports are missing. Second, loading of tomatoes does not go together well with the unloading of apples later. This could be a reporting inconsistency or a reporting incompleteness if indeed apples were (also) loaded earlier.

All these plausible explanations are provided by (E_1, I_1), (E_2, I_2), and (E_3, I_3), the three possible models of E_{init} and the program in Fig. 1. For space reasons we list only their EDB components, which are as follows:

E_1	E_2	E_3
$\mathsf{Load}(10, \mathsf{tomatoes}, \mathsf{pallet})$	$\mathsf{Load}(10, \mathsf{tomatoes}, \mathsf{pallet})$	$\mathsf{Load}(10, \mathsf{apples}, \mathsf{pallet})$
	$\mathsf{Load}(10, \mathsf{apples}, \mathsf{pallet})$	
$\mathsf{Unload}(45, \mathsf{container}, \mathsf{ship})$	$\mathsf{Unload}(45, \mathsf{container}, \mathsf{ship})$	$\mathsf{Unload}(45, \mathsf{container}, \mathsf{ship})$
$\mathsf{Unload}(50, \mathsf{pallet}, \mathsf{container})$	$\mathsf{Unload}(50, \mathsf{pallet}, \mathsf{container})$	$\mathsf{Unload}(50, \mathsf{pallet}, \mathsf{container})$
$\mathsf{Unload}(60, \mathsf{tomatoes}, \mathsf{pallet})$	$\mathsf{Unload}(60, \mathsf{apples}, \mathsf{pallet})$	$\mathsf{Unload}(60, \mathsf{apples}, \mathsf{pallet})$

In each of these models, the missing unloading events $\mathsf{Unload}(45, \mathsf{container}, \mathsf{ship})$ and $\mathsf{Unload}(50, \mathsf{pallet}, \mathsf{container})$ are added by repeated application of rule (4). Generally speaking, rule (4) inserts an Unload of the "containing container" the object to be unloaded from is in. The rules (7) – (9) all fix the "unloading apples vs. loading tomatoes" problem. Rule (7) leads to (E_1, I_1), rule (8) leads to (E_2, I_2), and rule (9) leads to (E_3, I_3). Each of these rules tests whether an object (apples) is swappable with another object (tomatoes) for the purpose of model revision, which is the case if the $\mathsf{SameBatch}$ relation says so. Notice that if E_{init} had, say $\mathsf{Unload}(60, \mathsf{oranges}, \mathsf{pallet})$ instead of $\mathsf{Unload}(60, \mathsf{apples}, \mathsf{pallet})$ then none of the rules (7) – (9) is applicable and no possible model exists. $\square$

5 Model Computation

This section introduces our calculus for computing possible models of stratified programs. It borrows some terminology from tableau calculi. A *path* p is a triple

(E, I, t) where E is an EDB, I is an IDB and $t \in \mathbb{N}$ is a timestamp. Intuitively, p represents the interpretation $\mathcal{I}((E \cup I)_{\leq t})$. An *initial path* is of the form $(E, \emptyset, 0)$. A *tableau* is a finite set of paths.[2]

Let $B = b_1, \ldots, b_k, \textbf{not}\, \vec{b}_{k+1}, \ldots, \textbf{not}\, \vec{b}_n$ be the body of a variable free stratified rule and A a set of ground ordinary atoms. A substitution σ with $dom(\sigma) = var(b_1, \ldots, b_k)$ is a *body matcher for B on A*, written as $(B, \sigma) \sqsubseteq A$, if the following holds:

(i) $\{b_1\sigma, \ldots, b_k\sigma\} \subseteq A \cup \mathrm{Th}(\Sigma_{\mathbb{N}})$, and
(ii) for no $i = k+1 \ldots n$ there is a grounding substitution γ for $\vec{b}_i\sigma$ such that $\vec{b}_i\sigma\gamma \subseteq A \cup \mathrm{Th}(\Sigma_{\mathbb{N}})$.

Note 1 (Computing body matchers). The definition of body matchers only applies to bodies of stratified rules. It is easy to see that a body matcher σ, if any exists, can be found by computing a simultaneous matching substitution σ for the ordinary atoms among $b_1, \ldots, b_k$ to A. Similarly for the substitution γ in condition (ii). Furthermore, stratification guarantees that all arithmetic atoms for testing conditions (i) and (ii) are necessarily ground and hence can be evaluated. $\square$

An inference rule is a schematic expression of the form $p \Rightarrow p_1, \ldots, p_k$ where p and p_j are paths, for all $1 \leq j \leq k$, where $k \geq 0$. It means that the premise p is to be replaced by the conclusions $p_1, \ldots, p_k$. An *inference* is an instance of an inference rule.

In the following, P is a stratified program and σ is a substitution such that $r\sigma$ is a variable free instance of a rule r that is clear from the context.

Ext: $(E, I, t) \;\Rightarrow\; (E, I \cup \mathcal{H}_1, t),\; \ldots,\; (E, I \cup \mathcal{H}_k, t)$
 if P contains an ordinary rule $h_1 \vee \cdots \vee h_k \leftarrow B$ such that
 $\{\mathcal{H}_1, \ldots, \mathcal{H}_k\} = \{\mathcal{H} \mid (B, \sigma) \sqsubseteq (E \cup I)_{\leq t} \text{ and } \emptyset \subsetneq \mathcal{H} \subseteq \{h_1\sigma, \ldots, h_k\sigma\}\}$
Restart: $(E, I, t) \;\Rightarrow\; (upd(E, \vec{e}_1), \emptyset, 0),\; \ldots,\; (upd(E, \vec{e}_k), \emptyset, 0)$
 if $k \geq 1$ and $\{\vec{e}_1, \ldots, \vec{e}_k\} = \{\vec{e}\sigma \mid \textbf{fail}(\vec{e}) \leftarrow B \in P,\ \vec{e} \neq \varepsilon \text{ and } (B, \sigma) \sqsubseteq (E \cup I)_{\leq t}\}$.
Fail: $(E, I, t) \;\Rightarrow$
 if P contains a rule $\textbf{fail}() \leftarrow B$ and there is a σ such that $(B, \sigma) \sqsubseteq (E \cup I)_{\leq t}$.
Jump: $(E, I, t) \;\Rightarrow\; (E, I, s)$ if s is the least timestamp in E with $t < s$.

The Ext rule extends I to satisfy all split rules for each case $\mathcal{H}$ of some instance of an ordinary rule in P whose body is satisfied by $(E \cup I)_{\leq t}$; Restart replaces the current path with *all* initial paths as per the non-empty fail rules after Ext is exhausted; Fail also terminates the current path but is to be applied

[2] This terminology is inspired by visualizing a set of paths as a tableau in the usual sense. For that, a path (E, I, t) leads to a branch whose nodes are labeled with the atoms $E \cup I$ and the branch as a whole is labeled with t. Moreover, the way the calculus constructs these paths sets indeed corresponds to a typical tableau construction. See, e.g., [3].

only if Restart doesn't; Jump advances the current time bound t. The following formalizes this intuition.

An initial path $(E, \emptyset, 0)$ is *new wrt. a tableau* T iff there is no I and no t such that $(E, I, t) \in T$. Let E_{init} be an *input EDB*. A *derivation* D *(from* E_{init} *and* P*)* is a sequence $(T)_{i \geq 0}$ of tableaus $D = (T_0 = \{(E_{\mathsf{init}}, \emptyset, 0)\}), T_1, T_2, \ldots$ such that, for all $i \geq 0$, there is a *selected path* $p \in T_i$ and $T_{i+1} = (T_i \setminus \{p\}) \cup \{p_1, \ldots, p_k\}$ where:

(a) $p \Rightarrow p_1, \ldots, p_k$ by Ext and $\{p_1, \ldots, p_k\} \not\subseteq T_i$,
(b) $p \Rightarrow q_1, \ldots, q_m$ by Restart and $\{p_1, \ldots, p_k\} = \{p \in \{q_1, \ldots, q_m\} \mid p \text{ is new wrt. } T_i\}$,
(c) $p \Rightarrow$ by Fail and $k = 0$, or
(d) $p \Rightarrow p_1$ by Jump and $k = 1$.

In addition, the inference rules must be prioritized in this order. That is, if T_{i+1} is obtained from T_i by, say, case (c) , then there is no tableau that can be obtained from T_i by case (a) or case (b) with the same selected path p; analogously for the other cases.

The derivation D is *exhausted* if it is finite and no inference rule is applicable to its final tableau T_n, for no $p \in T_n$. In this case the *computed models of* D is the set $\mathcal{M}(D) = \{(E, I) \mid (E, I, t) \in T_n \text{ for some } t \in \mathbb{N}\}$.

Figure 2 is a graphical illustration of a derivation and its computed models.

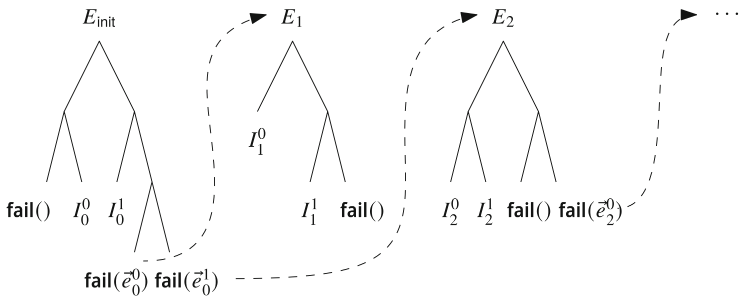

Fig. 2. Illustration of a hypothetical derivation. The root of each sub-tableau is labeled with the EDB in that sub-derivation. The first sub-tableau has two Restart inferences, leading to the second and third sub-tableau, where $E_1 = upd(E_{\mathsf{init}}, \vec{e}_0^0)$, $E_2 = upd(E_{\mathsf{init}}, \vec{e}_0^1)$. The isolated **fail**()s do not cause a Restart, they cause Fail. The computed models are $(E_{\mathsf{init}}, I_0^0)$, $(E_{\mathsf{init}}, I_0^1)$, (E_1, I_1^0), etc.

Theorem 1 (Soundness and completeness). *Assume a signature* Σ *without* k*-ary function symbol, for* $k > 0$*. Let* P *be a stratified program and* E_{init} *an EDB. Assume an exhausted derivation* D *from* E_{init} *and* P*. Then* $\mathcal{M}(D) = mods_P(E_{\mathsf{init}})$*.*

Proof. (Sketch) Let T_n be the final tableau of D. For soundness, assume $\mathcal{M}(D) \neq \emptyset$ and chose any $(E, I) \in \mathcal{M}(D)$ arbitrary. That is, $(E, I, t) \in T_n$, for some t. We have to show $(E, I) \in \mathit{mods}_P(E_{\mathsf{init}})$, equivalently $(E_{\mathsf{init}}, E, I) \models P$.

The EDB E is either E_{init} or derived from E_{init} through, say, $k > 0$ intermediate EDBs by Restarts. By induction on k one can show that, on the semantic side, E is a restart induced by P and E_{init}, i.e., $E \in \mathcal{E}$ in Definition 5. This follows from the definition of derivations. In particular, the earliest-time requirement in Definition 5 is matched by prioritizing Restart over Fail and Jump.

With the EDB E traced down in $\mathcal{E}$, it remains to prove $(E, I) \models P$. With the stratification of P (Definition 1) this is rather straightforward. Range-restrictedness makes sure that only ground heads are derivable. The Ext inference rule achieves on-the-fly splitting and only for those variable-free instances of rules whose body is satisfied, which are the only ones that count (details in [26,27]). The requirements on EDB/IDB atoms in negative body literals ($\leq$ vs. $<$) in Definition 1 entail that utilizing body matchers in derivations is correct wrt. the rule semantics in Definition 2.

The timed setting requires a layered fixpoint iteration. Stratification makes sure that stepwisely incrementing in derivations the time bound by Jump inferences is all that is needed to comply with the "unstepped" possible model semantics in Definitions 4 and 5. In particular, no ordinary rule can derive in its head a conclusion with a timestamp earlier than the latest timestamp in its body. This makes the derivability relation monotonic wrt. increasing time stamps (usual stratification by predicates is covered in [26]). Moreover, because the given derivation is exhausted, no fixpoint iteration can stop prematurely.

For completeness, assume $\mathit{mods}_P(E_{\mathsf{init}}) \neq \emptyset$ and chose any $(E, I) \in \mathit{mods}_P(E_{\mathsf{init}})$ arbitrary. We have to show $(E, I) \in \mathcal{M}(D)$. The first step is to locate in D the sub-tableau with E at its root, by tracing $E \in \mathcal{E}$ from E_{init}. The next step then is to argue for the completeness of the sub-tableau construction with that fixed E, giving $(E, I) \in T_n$. All that uses similar considerations as the soundness proof above.

One important detail is that Ext is the highest-priority inference rule. This makes sure that no model candidate is terminated too early, so that all possible branching out takes place. As a consequence, *all* possible fail heads for the current time point will be derived. The requirement that there are no proper (non-constant) function symbols make sure that the layered fixpoint computation of derivations terminates and finds (E, I). □

6 Implementation

It is not too difficult to translate the model computation calculus of Sect. 5 into a proof procedure. Tableaux can be represented in a direct way, as a set of paths (E, I, t). In terms of Fig. 2, the proof procedure can implement a one-branch-at-a-time approach for one sub-tableau at a time, for space efficiency, embedded in an adapted given-clause loop algorithm. Only the EDBs, not the full models, need to be remembered to implement the "Progress" condition for derivations (cf. Sect. 5).

A concrete implementation could based on, e.g., hyper resolution with splitting [5] or hyper tableaux [3] calculi. Our implementation however is implemented in an unusual way, by shallow embedding in Scala.[3]

Scala [28] is a modern high-level programming language that combines object-oriented and functional programming styles. It has functions as first-class objects and supports user-definable pre-, post- and infix syntax. With these features, Scala is suitable as a host language for embedding domain-specific languages (DSLs). (See, e.g., [16] for a Scala DSL for runtime verification.) In our case, the logic program rules are nothing but partial functions, instantiating and evaluating a rule body reduces to partial function definedness, and deriving a rule head reduces to executing a partial function on a defined point. An advantage of the DSL approach is that is is easy to interface with external systems, e.g., databases, in particular if they have a Java interface.

Moreover, it is easy to make the full Scala language and its associated data structure libraries available for writing rules. (There is no theoretical problem doing that as long as all Scala expressions are ground and, hence, can be evaluated.) For example, the EDB in Example 3 has the Scala-set forming term Set(tomatoes, apples), and the rules (7) - (9) in Fig. 1 test in their last lines membership in such sets by Scala expressions.

While EDBs are naturally written as Scala source code, logic program rules are usually written in a (much) more convenient syntax and translated into the required format by the Scala macro mechanism. See Listing 1.1 for an example.[4]

Listing 1.1. Sample EDB/IDB declarations and a rule. Some unimportant declarations left away.

```
1 type Time = Int
2 case class Load(time: Time, obj: String, cont: String) extends EDBAtom
3 case class SameBatch(time: Time, objs: Set[String]) extends EDBAtom
4 case class In(time: Time, obj: String, cont: String) extends IDBAtom
5 @rules
6 val rules = List(In(time, obj, cont) :- (In(time, obj, c), In(time, c, cont))
```

In Listing 1.1, line 1 sets the concrete type for time to Int, the Scala integers. A realistic application could use a rich time class like java.time.OffsetDataTime. Lines 2, 3 and 4 define the EDB and IDB signature of the supply chain Example 2 by extension of the Scala classes EDBAtom and IDBAtom. The Load relation says that the object obj was loaded into the container cont at time time. The In relation says that obj was in container cont at time time. The *time* argument must be named time, but all other arguments and their types can be freely chosen. For simplicity we used strings, except for the SameBatch relation, which has a set of strings for its objs parameter.

Line 6 defines a list with one rule that expresses the transitivity of the In relation (rule (2) in Fig. 1). Line 5 is an annotation that tells the compiler to

[3] Our implementation, the Fusemate system, is available at https://bitbucket.csiro.au/users/bau050/repos/fusemate/.

[4] Scala case classes are records in language agnostic terminology.

expand the subsequent definition by a macro named rules. Indeed, without the help of a macro the rule would not compile because of undefined variables in the rule. The macro expansion of the rule in line 6 is the Scala function in Listing 1.2.

Listing 1.2. Macro expansion of the transitivity rule.

```
(I: Set[Lit]) => {
  case List(In(time, obj, c), In(time0, c1, cont)) if
    c == c1 && time == time0 => In(time, obj, cont)
  }
```

The anonymous function in Listing 1.2 is passed in its formal parameter I a set of atoms which will always be the "current interpretation" $(E \cup I)_{\leq t}$ where $p = (E, I, t)$ is the current path. The set I is needed for evaluating negative body literals and is not relevant for this example. The function returns a partial function in the form of a case expression. The pattern in the case expression are the *ordinary* atom of the *positive* body literals. These are the ones that need to be matched to the atoms of $(E \cup I)_{\leq t}$ for rule evaluation (see Note 1). The matching is done by applying the partial function to all tuples (of the proper arity) of elements from the current interpretation. If the application succeeds, i.e., if the case pattern match succeeds and the additional if-condition is satisfied, the partial function body (to the right of =>) is executed, which results in an instantiated head. If the head is disjunctive, all non-empty subsets are taken. This gives all split programs (cf. Definition 3) on the fly. The resulting sets are collected in one sweep for each rule and are candidates for extending the current path. **fail**-rules are processed in a similar way.

Notice that the pattern of a case expression needs to be linear, hence the renaming apart of pattern variables and the obvious equalities in the if-condition. Notice also that substitutions are not explicitly represented, they are hidden in the Scala runtime system.

It remains to be explained how arithmetic atoms and negative body literals are macro-expanded. By way of example, consider, say, rule (7) in Fig. 1. Its arithmetic condition $t < \mathit{time}$ is simply conjoined to the **if**-condition of the rule's case expression. An **if**-expression like **if**((b contains obj) and (b contains o)) is a backdoor for adding arbitrary Boolean-valued Scala code to that condition ("contains" belongs to the Scala library and tests set membership). Recall that all variables in arithmetic atoms will always be ground instantiated by matching and hence can be evaluated. This must be be extended to Scala conditions. In the implementation, any such free variable would be detected as a compile time error.

The body literal **not**(load(t, obj, cont), t < time) is expanded into a partial function **case** load(t, obj1, cont1) **if** obj == obj1 && cont ==cont1 && t < time => Abort. It is structurally the same as the one in Listing 1.1 except that the binding of the variables obj and cont in the surrounding context need to respected, giving the stated equalities. Now, the **if**-condition of the case expression of the surrounding rule (i.e., rule 7) is conjoined with a Scala expression for testing that the partial function does not return Abort, for no tuple of elements from the set I explained above.

The implementation supports some more features not further discussed here: rules can be defined locally within case classes; literals – possibly negated atoms – can be used anywhere instead of atoms; a "strong fail" head operator terminates a model candidate without restarts, e.g., for classical negation: a and $\neg a$ together is unfixable; and Scala conditions can access the interpretation $(E \cup I)_{\leq t}$ for, e.g., concise data aggregation.

7 Conclusions

This paper presented a novel calculus and implementation for situational awareness applications. The approach is meant to be *practical* in *three* ways: first, realistic situational awareness requires being able to reason with incomplete or erroneous data. Moreover, "anytime" reasoning is needed, meaning that a model can be derived, rejected or repaired at any current time. Our approach supports these needs with a (disjunctive) logic programming framework with timed predicates, stratified negation and a novel model revision operator. Second, thanks to implementation on top of Scala, it is trivial to attach arbitrary Scala code and Java libraries. (It would not be difficult to extend the calculus respectively.) For instance, reading in XML data and making them available as terms (Scala case classes) is easy. Third, we strived for a "cheap" model computation procedure that makes do without additional generate-and-test needs. As such, it is perhaps more adequately seen as study in pushing bottom-up first-order logic model computation technology rather than slimmed down answer-set programming.

As for future work, one interesting idea is to add probabilities to the picture, say, in the way ProbLog extends Prolog. This is obviously useful because, e.g., some explanations (models) or repairs (restarts) are more likely than others. Another idea is to view the model computation as runtime verification. This view suggests that (probabilistic) linear temporal logic could serve as an additional useful high-level specification language component.

Acknowledgements. I thank the reviewers for their constructive comments. Yuzhou Chen discovered an error in an earlier version of the paper.

References

1. Alchourròn, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: partial meet contraction and revision functions. J. Symb. Logic **50**, 510–530 (1985)
2. Baader, F., et al.: A novel architecture for situation awareness systems. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS (LNAI), vol. 5607, pp. 77–92. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02716-1_7
3. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper tableaux. In: Alferes, J.J., Pereira, L.M., Orlowska, E. (eds.) JELIA 1996. LNCS, vol. 1126, pp. 1–17. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61630-6_1

4. Baumgartner, P., Furbach, U., Pelzer, B.: Hyper tableaux with equality. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 492–507. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_36
5. Baumgartner, P., Schmidt, R.: Blocking and other enhancements for bottom-up model generation methods. J. Autom. Reason. **64**, 197–251 (2019). https://doi.org/10.1007/s10817-019-09515-1
6. Bradley, A., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8
7. Chesani, F., Mello, P., Montali, M., Riguzzi, F., Sebastianis, M., Storari, S.: Checking compliance of execution traces to business rules. In: Ardagna, D., Mecella, M., Yang, J. (eds.) BPM 2008. LNBIP, vol. 17, pp. 134–145. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00328-8_13
8. Cordier, M., Thiébaux, S.: Event-based diagnosis for evolutive systems. In: Proceedings of 5th International Workshop on Principles of Diagnosis, pp. 64–69 (1994)
9. De Giacomo, G., Maggi, F.M., Marrella, A., Sardina, S.: Computing trace alignment against declarative process models through planning. In: Proceedings of 26th International Conference on Automated Planning and Scheduling (ICAPS) (2016)
10. Eiter, T., Gottlob, G.: Complexity results for disjunctive logic programming and application to nonmonotonic logics. In: Proceedings of the 1993 International Symposium on Logic Programming, ILPS 1993, pp. 266–278. MIT Press, Cambridge (1993)
11. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**, 364–418 (2001)
12. Endsley, M.: Toward a theory of situation awareness in dynamic systems. Hum. Factors J.: J. Hum. Factors Ergon. Soc. **37**(1), 32–64 (1995)
13. Gelfond, M.: Answer sets. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 285–316. Elsevier, Amsterdam (2008)
14. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the 5th International Conference on Logic Programming, Seattle, pp. 1070–1080 (1988)
15. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Gener. Comput. **9**, 365–385 (1991). https://doi.org/10.1007/BF03037169
16. Havelund, K., Joshi, R.: Modeling rover communication using hierarchical state machines with scala. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10489, pp. 447–461. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66284-8_38
17. Kauffman, S., Havelund, K., Joshi, R.: `nfer` – a notation and system for inferring event stream abstractions. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 235–250. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_15
18. Lifschitz, V.: Action languages, answer sets, and planning. In: Apt, K.R., Marek, V.W., Truszczynski, M., Warren, D.S. (eds.) The Logic Programming Paradigm: A 25-Year Perspective, pp. 357–373. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-642-60085-2_16
19. Marek, V.W., Truszczyński, M.: Revision specifications by means of programs. In: MacNish, C., Pearce, D., Pereira, L.M. (eds.) JELIA 1994. LNCS, vol. 838, pp. 122–136. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0021968
20. Marek, V.W., Truszczyński, M.: Revision programming. Theoret. Comput. Sci. **190**(2), 241–277 (1998)

21. McIlraith, S.: Toward a theory of diagnosis, testing and repair. In: Proceedings of 5th International Workshop on Principles of Diagnosis, pp. 185–192 (1994)
22. Niemelä, I.: A tableau calculus for minimal model reasoning. In: Miglioli, P., Moscato, U., Mundici, D., Ornaghi, M. (eds.) TABLEAUX 1996. LNCS, vol. 1071, pp. 278–294. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61208-4_18
23. Pelzer, B., Wernhard, C.: System description: E- KRHyper. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 508–513. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_37
24. Peppas, P.: Chapter 8 belief revision. In: van Harmelen, F., Lifschitz, V., Porter, B. (eds.) Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 317–359. Elsevier, Amsterdam (2008)
25. 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, Burlington (1988)
26. 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 (1990)
27. Sakama, C., Inoue, K.: An alternative approach to the semantics of disjunctive logic programs and deductive databases. J. Autom. Reason. **13**, 145–172 (1994). https://doi.org/10.1007/BF00881915
28. The Scala Programming Language. https://www.scala-lang.org
29. Schwind, N., Inoue, K.: Characterization of logic program revision as an extension of propositional revision. Theory Pract. Logic Program. **16**(1), 111–138 (2016)
30. Zaniolo, C.: Expressing and supporting efficiently greedy algorithms as locally stratified logic programs. Technical Communications of ICLP 2015 1433 (2015)