A Logic-Based Framework for Reactive Systems

Robert Kowalski and Fariba Sadri

Department of Computing
Imperial College London
{rak,fs@doc.ic.ac.uk}

Abstract. We sketch a logic-based framework in which computation consists of performing actions to generate a sequence of states, with the purpose of making a set of reactive rules in the logical form *antecedents* → *consequents* all true. The *antecedents* of the rules are conjunctions of past or present conditions and events, and the *consequents* of the rules are disjunctions of conjunctions of future conditions and actions. The *antecedents* can be viewed as complex/composite events, and the *consequents* as complex/composite/macro actions or processes.

States are represented by sets of atomic sentences, and can be viewed as global variables, relational databases, Herbrand models, or mental representations of the real world. Events, including actions, transform one state into another. The operational semantics maintains only a single, destructively updated current state, whereas the model-theoretic semantics treats the entire sequence of states, events and actions as a single model. The model-theoretic semantics can be viewed as the problem of generating a model that makes all the reactive rules true.

Keywords: reactive systems, model generation, LPS, KELPS, complex events, complex actions.

1 Introduction

Reactive rules in one form or another play an important role in many different areas of Computing. They are explicit in the form of condition-action rules in production systems, event-condition-action rules in active databases, and plans in BDI agents. Moreover, they are implicit in many other areas of Computing.

David Harel [7], for example, identifies reactive systems as one of the main kinds of computational system, characterizing them as having the general form "when event α occurs in state ***A***, if condition C is true at the time, the system transfers to state B". Harel [8] notes that StateCharts, a graphical language for specifying reactive systems, is "the heart of the UML - what many people refer to as its driving behavioral kernel".

Reactive systems can be regarded as an extension of transition systems. As Wolfgang Reisig [22] puts it, an initialized, deterministic transition system is "a triple C = (Q, I, F) where Q is a set (its elements are denoted as states), I ⊆ Q (the initial states), and F : Q → Q (the next-state function)". Transition systems can be extended to reactive systems in which the transition from one state to the next is "not conducted by the program, but by the outside world".

But even in their simpler "initialized, deterministic" form, transition systems have been proposed as a general model of Computing. Reisig points out that, in the first volume of *The Art of Computer Programming* [12], Donald Knuth suggests their use as a general semantics for algorithms.

In this paper, we propose a simplified kernel, KELPS, of the Logic-based agent and Production System language LPS [15, 16, 17]. Whereas LPS combines reactive rules and logic programs, KELPS consists of reactive rules alone. The distinguishing features of LPS and KELPS are that:

- They interpret reactive rules of the form *antecedents* $\rightarrow$ *consequents* as universally quantified sentences in first-order logic (FOL).
- They have an operational semantics in which actions and other events destructively update a current state.
- They have a model-theoretic semantics in which actions are performed to make the reactive rules all true in a single classical model associated with the sequence of states and events.

The aim of this paper is to highlight the significance of reactive rules in LPS, and to take advantage of the simplicity of KELPS to introduce some extensions to LPS. The main extensions are the addition of complex events in the antecedents of reactive rules, the generalization of the sequential ordering of conditions and events to partial ordering, and the representation of conditions by arbitrary formulas of FOL.

The paper is structured as follows: Section 2 introduces KELPS by means of an example. Section 3 sketches the background of LPS and KELPS. Sections 4-5 describe the syntax, model-theoretic and operational semantics of KELPS; and section 6 discusses soundness and incompleteness. Section 7 describes the extension of KELPS by logic programs; and section 8 sketches an extension to the multi-agent case. Sections 9 and 10 conclude.

2 Example

We use a variant of an example in [9]. In this variant, a reactive agent monitors a building for any outbreaks of fire. The agent receives inputs from a fire alarm pre-sensor and a smoke detector. If the agent detects a possible fire, then it activates local fire suppression devices and in the case that there is no longer a possible fire, calls for a security guard to inspect the area. Alternatively, it calls the fire department. This behaviour can be represented in KELPS by means of a reactive rule such as:

if pre-sensor detects possible fire in area A at time T_1
and smoke detector detects smoke in area A at time T_2
and $|T_1 - T_2| \leq 60$ *sec and max*(T_1, T_2, T)
then [[activate local fire suppression in area A at time T_3 *and* $T < T_3 \leq T + 10$ *sec*
and not fire in area A at time T_4 *and* $T_3 < T_4 \leq T_3 + 30$ *sec*
and send security guard to area A at time T_5 *and* $T_4 < T_5 \leq T_4 + 10$ *sec]*
or call fire department to area A at time T_3' *and* $T < T_3' \leq T + 60$ *sec]*

Here A, T_1, T_2 and T are implicitly universally quantified with scope the entire rule, but T_3, T_3', T_4 and T_5 are implicitly existentially quantified with scope the consequents of the rule.

The notation employed in this example is an informal version of the internal syntax of KELPS with an explicit representation of time. It is compatible with a variety of external notations that hide the representation of time. We employ this internal syntax throughout the paper, as it clarifies the operational and model-theoretic semantics.

Notice that the antecedents of the rule represent a complex event and the consequents represent a complex action consisting of two alternative plans. The first plan consists of two actions and a condition, and the second consists of an action. Both plans include temporal constraints. In practice, the first plan might be preferred and tried before the second. If any part of the first plan fails, then the second plan can be tried. Moreover, even if some of the first plan fails, it can be retried as long as the temporal constraints can be satisfied. If both plans fail and cannot be retried, then the reactive rule cannot be made true. This can be avoided by adding additional alternative plans to the consequents of the rule.

3 LPS and KELPS in Context

LPS is a direct descendant of ALP agents [14], which descend in turn from abductive logic programming (ALP). ALP extends logic programs with undefined (open or abducible) predicates, which are constrained by integrity constraints [11]. ALP agents embed ALP in an agent cycle, in which logic programs serve as the agent's *beliefs*, and integrity constraints serve as the agent's *goals*.

In ALP agents, the world is represented, as in the situation calculus [20], event calculus [19] and other knowledge representation schemes in AI, by atomic predicates with time or state parameters, called *fluents*. Updates of the world are described by an *event theory* that specifies the fluents that are initiated and terminated by events. *Frame axioms* of one form or another express the property that if a fluent holds in a state, then it continues to hold in future states unless and until it is terminated by an event.

Frame axioms used to reason about states of affairs exact a heavy computational penalty, which is prohibitively large for even moderately sized knowledge bases. As a consequence, frame axioms are rarely used in practical applications.

ALP agents use the event calculus with its frame axiom to reason about fluents. However, ALP agents also have the option of observing fluents, instead of reasoning to derive them. This is one step towards eliminating frame axioms entirely. The next and final step, which historically resulted in LPS, came from trying to simulate production systems by means of ALP agents.

This final step was made difficult by the overabundance of semantics for ALP. The operational semantics of ALP agents, in particular, is based on the IFF proof procedure [6], which is complete for the Kunen three-valued semantics. The attractiveness of this completeness result was an obstacle to identifying a semantics that is better suited for production systems. Eventually it became apparent that the necessary semantics is one of the simplest possible, namely the minimal model semantics of Horn clauses [3].

In LPS, an agent's beliefs are represented by logic programs, and the agent's goals are represented by reactive rules, which are a special case of integrity constraints in ALP. The purpose of the agent's actions is to extend its beliefs about the world, so that its reactive rules are all true in the minimal model of the extended beliefs. The minimal model is unique if the beliefs are represented by Horn clauses [3]. If the beliefs are represented by locally stratified logic programs [21], then the extended beliefs have a unique intended, minimal model, called the perfect model.

In retrospect, KELPS is probably closest to the modal agent language MetateM. In MetateM a program consists of sentences in logical form:

'past and present formula' **implies** 'present or future formula' [4]

and computation consists of generating a model in which all such sentences are true. These sentences are represented in a modal temporal logic, with truth values defined relative to a possible world embedded in a collection of possible worlds connected by a temporal accessibility relation. In contrast, in KELPS, similar sentences are expressed in classical FOL with an explicit representation of time, with truth values defined relative to a single classical model, in the standard Tarskian manner. Moreover, whereas MetateM uses frame axioms to reason about possible worlds, KELPS uses destructive updates to transform one state into another.

In other respects, full LPS is closer to Transaction Logic ($\mathcal{TR}$ Logic) [1], which also maintains only the current state, using destructive updates without frame axioms. Moreover, like LPS and MetateM, $\mathcal{TR}$ Logic also gives a model-theoretic semantics to complex actions (and transactions), by interpreting them as sentences in logical form and by generating a model in which these sentences are true.

However, in $\mathcal{TR}$ Logic, transactions are expressed in a non-standard logic with logical connectives representing temporal sequence. The model in $\mathcal{TR}$ Logic is a collection of possible worlds, as in modal logic. But truth values are defined relative to paths between possible worlds.

In LPS, in contrast with both MetateM and $\mathcal{TR}$ Logic, truth values are defined relative to a single, classical model, which contains the entire sequence of states and state-transforming events. Possible worlds in the semantics of MetateM and $\mathcal{TR}$ Logic become sub-models of this single model in LPS, and the accessibility relation becomes a relation expressed by means of event descriptions and temporal constraints represented explicitly in the language. This simplifies the semantics and increases the expressive power of the language by making events and times first class objects.

4 KELPS and its Model-Theoretic Semantics

Viewed in agent-oriented terms, *reactive rules* in KELPS are *maintenance goals* expressed in logical form. Their truth values are determined by the agent's *beliefs*, which include an initialised sequence of states $\boldsymbol{S_0}, \boldsymbol{S_1},\ldots, \boldsymbol{S_i},\ldots$ and an associated sequence of state-transforming sets of events $\boldsymbol{e_1},\ldots, \boldsymbol{e_i},\ldots$

States $\boldsymbol{S_i}$ in KELPS are sets of atomic sentences, also called *facts* or *fluents*. They can be understood as representing sets of global variables, relational databases,

Herbrand models, or mental representations of the real world. *Events* $\boldsymbol{e_i}$ are also sets of atomic sentences, and represent either external events or the agent's own actions.

4.1 The syntax of reactive rules

In the model-theoretic semantics, facts, events and reactive rules are represented with time-stamps. However, in the operational semantics, facts are represented without time-stamps, so that facts that are not affected by an event simply persist from one state to the next without needing to reason with frame axioms that they persist.

Correspondingly, reactive rules can also be represented either in an internal syntax with time parameters, or in an external syntax without explicit time. The internal syntax underpins the model-theoretic semantics, and is consequently more basic. It also facilitates more flexible and more powerful ways of representing and reasoning with temporal constraints. For these reasons we focus on the internal syntax in this paper.

At the expense of restricting its expressive power, the internal syntax is compatible with a variety of external notations. In [16, 17], we specified an external syntax in which temporal ordering is indicated by the order in which formulas are written and by special logical connectives. However, the internal syntax is also compatible with other external notations, such as the syntax of $\mathcal{TR}$ Logic, in which $P \otimes Q$ means "do P and then do Q". In LPS/KELPS this translates into $P(T_1) \wedge Q(T_2) \wedge T_1 < T_2$.

The internal syntax is also compatible with a modal external syntax. For example, $P \wedge \Diamond Q$ can also be translated into $P(T_1) \wedge Q(T_2) \wedge T_1 < T_2$. Graphical notations for partial ordering are also possible.

Here we define the *internal syntax* of the reactive rules. A *reactive rule* (or more simply, a *rule*) is a universally quantified conditional sentence of the form[1]:

$$\forall X\ [antecedents(X) \rightarrow \exists Y\ consequents(X, Y)]$$

X is the set of all time variables and any other unbound variables that occur in *antecedents*, and Y is the set of all time variables and any other unbound variables that occur only in the *consequents*. This convention allows us to omit the explicit representation of these two quantifiers and write rules in the simpler form:

$$antecedents(X) \rightarrow consequents(X, Y)$$

To ensure that the actions and conditions in *consequents(X, Y)* occur after the events and conditions in *antecedents(X)*, the time variables in Y should be constrained directly or indirectly in *consequents* to be later than the time variables in X.

Antecedents of rules can be viewed as complex (or composite) events, and *consequents* can be viewed as complex actions (or processes). The *antecedents* are a conjunction, each conjunct of which is either a condition, an event atom or a temporal constraint, where:

[1] This notation means that X includes all the variables that occur in the *antecedents(X)*. But it does not mean that *consequents(X, Y)* contains all of the variables in X.

- A *condition* is an FOL formula including only a single time parameter in the vocabulary of the state predicates, possibly including auxiliary state-independent predicates that do not change with time.[2]
- An *event atom* is an atomic formula with a single time parameter representing the occurrence of an event. Similarly an *action atom* is an event atom in which the event is an action.
- A temporal constraint is an inequality $time_1 < time_2$ or $time_1 \leq time_2$, where $time_1$ and $time_2$ represent time points, one of which is a variable, and the other of which is a variable or constant.[3]

The *consequents* of a rule are a disjunction, each disjunct of which is a conjunction, each conjunct of which is either a condition, an action atom or a temporal constraint.

Temporal constraints in the *antecedents* should involve only time variables occurring elsewhere in the *antecedents*, and temporal constraints in the *consequents* should involve only time variables occurring elsewhere in the rule.[4]

The various restrictions on the syntax described above simplify the operational semantics presented later. For example, these restrictions limit complex events to partially ordered sequences of conditions and simpler events. In particular, they do not allow complex events that include a constraint that no event of a certain kind occurs within a certain period of time.[5] Moreover, they do not allow external events in the consequents of rules.

Both of these restrictions can be removed at the expense of complicating the operational semantics. In fact, as we will see in the next section, the syntax of reactive rules can be generalized to include arbitrary sentences of FOL.

4.2 The model-theoretic semantics

Viewed in general terms, the task in KELPS is to generate a Herbrand model that makes a set of sentences in FOL all true. The core of the model is determined by a sequence of states $\boldsymbol{S_0}$, $\boldsymbol{S_1}$,…, $\boldsymbol{S_i}$,…. and state-transforming sets of events $\boldsymbol{e_1}$,…, $\boldsymbol{e_i}$,…. These sequences are combined into a single model by time-stamping facts and events.

The model-theoretic semantics is compatible with a variety of different notions of time and temporal ordering. However, for simplicity in this paper, we assume that time is discrete, and that events $\boldsymbol{e_i}$ are instantaneous, and are stamped with the time t_i of their occurrence, written either as $\boldsymbol{e_i}(t_i)$, $happens(\boldsymbol{e_i}, t_i)$ or simply $\boldsymbol{e_i}^*$.

We also assume that the time t_i at the beginning of a state $\boldsymbol{S_i}$ "lasts" until the time t_{i+1} at the end of state $\boldsymbol{S_i}$. So in effect $t = t_i$ for all $t_i \leq t < t_{i+1}$. The time-stamping of a

---

[2] For example, ***Aux*** might contain atomic sentences defining a predicate *area A is connected to area B*, which might be useful in a refinement of the reactive rule in section 2.

[3] We also need to allow arithmetic expressions such as $t + n$. This can be done by replacing such expressions by variables, and by including auxiliary conditions that perform the necessary arithmetic. E.g. replace $t + n$ by a variable T, and add a condition $plus(t, n, T)$.

[4] To simplify both the model-theoretic and operational semantics we need to impose a *range restriction* on conditions, preventing such rules as $\neg p(X, T_1) \rightarrow q(X, Y, T_2) \wedge T_1 < T_2$, in which the variable X is unrestricted.

[5] E.g. the complex event *stock goes up more than 5% at time T_1 and stock goes up more than 5% at a later time T_2 and stock does not go down more than 2% between T_1 and T_2.*

fact p that is true in a state $\boldsymbol{S_i}$ is written either as $p(t_i)$ or as $holds(p, t_i)$. The resulting state $\boldsymbol{S_i}$ in which all its facts have been time-stamped is written simply as $\boldsymbol{S_i^*}$.

Because the conditions and events in the *antecedents* and *consequents* of reactive rules are temporally ordered by inequality constraints between time points, the model-theoretic semantics needs a specification of the inequality relation. In the operational semantics, the inequality relation can be computed by any means that respects this specification, including the use of temporal constraint processing. However, in the model theory, the specification of the temporal ordering is represented extensionally by an infinite set of atomic sentences $t \leq t'$ and $t < t'$ for time points.

This extensional representation of the inequality relations is included in a set ***Aux***, which also includes extensional representations of any other auxiliary predicates, such as absolute value | | and *max* in the example of section 2. In LPS, these auxiliary predicates are defined by logic programs.

With this time-stamping of facts and events, and this extensional representation of the inequality relation and of any other auxiliary predicates, the sequence of states and events can be combined into a single Herbrand model:

$$\boldsymbol{M = Aux \cup S_0^* \cup e_1^* \cup S_1^* \cup e_2^* \cup \ \ldots \ \cup e_i^* \cup S_i^* \cup e_{i+1}^*} \ldots..$$

In general, a *Herbrand model*, which is a set of atomic sentences, has a dual interpretation. As a set of sentences, it is a purely syntactic object. But, as a specification of the set of all atomic sentences that are true in the *Herbrand universe* (i.e. the set of all variable-free terms that can be constructed from the vocabulary of the language), it is a model-theoretic structure. These two interpretations are closely related: A Herbrand model, viewed as a model-theoretic structure, is the *unique minimal model* of itself, viewed syntactically as a set of sentences. As we will see in section 7, this minimal model relationship is the key to the semantics of the more general case, in which the set ***Aux*** is generalized to a logic program.

The definition of truth in a Herbrand model for arbitrary sentences of FOL follows the classical Tarskian definition. This includes the definition of truth for reactive rules: A rule of the form $\forall X\ [antecedents(X) \rightarrow \exists Y\ consequents(X, Y)]$ is true in $\boldsymbol{M}$ if and only if, for every ground instance x over the Herbrand universe of the variables X, if $antecedents(x)$ is true in $\boldsymbol{M}$, then there exists a ground instance y over the Herbrand universe of the variables Y such that $consequents(x, y)$ is true in $\boldsymbol{M}$.

Notice that, in theory, because truth is defined for arbitrary sentences of FOL, the set of reactive rules could be replaced by any set of FOL sentences. The syntactic restrictions on reactive rules have been imposed primarily to simplify the operational semantics so that it operates with only a single current state.[6]

Given a set of reactive rules $\boldsymbol{R}$, an initial state $\boldsymbol{S_0}$ and a sequence of sets of external events $\boldsymbol{ex_1}, \ldots, \boldsymbol{ex_i}, \ldots$, the *task* in KELPS is to generate an associated sequence of sets of actions $\boldsymbol{a_1}, \ldots, \boldsymbol{a_i}, \ldots$ such that all of the reactive rules in $\boldsymbol{R}$ are true in the resulting Herbrand model $\boldsymbol{M = Aux \cup S_0^* \cup e_1^* \cup S_1^* \cup e_2^* \cup \ \ldots \ \cup e_i^* \cup S_i^* \cup e_{i+1}^*}$

[6] For example, in the operational semantics, only the last set of events $\boldsymbol{e_i}$ is accessible during the *i-th* cycle. This restriction can be liberalised to include a window of previous events, with the benefit that more complex events can be catered for without too much loss of efficiency.

….. Here $\boldsymbol{e_i} = \boldsymbol{ex_i} \cup \boldsymbol{a_i}$ and $\boldsymbol{S_{i+1}}$ is obtained from $\boldsymbol{S_i}$ by adding any fluents initiated by the events in $\boldsymbol{e_{i+1}}$ and by deleting any fluents terminated by the events in $\boldsymbol{e_{i+1}}$.

5 KELPS Operational Semantics

The model-theoretic semantics of KELPS is compatible with many different operational semantics. The operational semantics sketched in this section uses the internal syntax, and employs an observe-think-decide-act agent cycle. It is relatively abstract, and is itself compatible with many different implementations.

We assume that the *i-th* cycle starts at time t_i, when the events $\boldsymbol{e_i}$ transform the state $\boldsymbol{S_{i-1}}$ into the state $\boldsymbol{S_i}$, and that it ends at time $t_{i+1} = t_i + \varepsilon_i$, where ε_i is the duration of the cycle. We assume that this duration ε_i is short enough that actions can be executed in a timely manner, and long enough that all the relevant conditions can be evaluated in the current state within a single cycle. In addition, we assume that the state $\boldsymbol{S_i}$ remains unchanged throughout the period from t_i to t_{i+1} i.e. $t = t_i$ for all $t_i \leq t < t_{i+1}$. This is so that the truth values of conditions evaluated in $\boldsymbol{S_i}$ are determined relative to a single time point t_i, but their evaluation can take place during the interval from t_i to t_{i+1}. Any events are observed and assimilated only at the beginning of cycles.

In addition to maintaining the current state $\boldsymbol{S_i}$, the operational semantics maintains a *goal state* $\boldsymbol{G_i}$, which is a set of *achievement goals*, each of which can be regarded as a set of alternative conditional plans of actions for the future. Logically each goal state is a conjunction, and each conjunct is a disjunction of existentially quantified conjunctions of temporally constrained conditions and actions. These achievement goals are typically derived from the *consequences* of reactive rules (maintenance goals). However, they can also be given separately in the initial goal state $\boldsymbol{G_0}$.

Operationally each achievement goal in the goal state can be regarded as a separate thread. In particular different threads, and different alternatives within the same thread, do not share any variables.[7]

To deal with complex events in the *antecedents* of reactive rules, the operational semantics also maintains a current set of reactive rules $\boldsymbol{R_i}$. A new rule is added to $\boldsymbol{R_i}$ when an instance of a condition or event in the *antecedents* of a rule is true in the current state. The new rule is an instance of the rest of the original rule that needs to be maintained in the future.

To simplify the operational semantics, and to avoid over-constraining times, we exclude conditions and events of the form $p(t)$ in reactive rules, where t is a time constant. Instead, we write $p(T) \wedge t \leq T \leq t + \delta$ for some suitably small δ.

With these simplifying assumptions, the operational semantics begins with an initial state $\boldsymbol{S_0}$, goal state $\boldsymbol{G_0}$ and set of reactive rules $\boldsymbol{R_0}$. The *i-th* cycle, starting with state $\boldsymbol{S_{i-1}}$, goal state $\boldsymbol{G_i}$, rules $\boldsymbol{R_i}$ and events $\boldsymbol{e_i}$ at time t_i consists of the following steps:

Step 0. Observe. State $\boldsymbol{S_{i-1}}$ is transformed into state $\boldsymbol{S_i}$ by adding and deleting all the facts associated with all the events in $\boldsymbol{e_i}$.

Step 1. React. For *every* reactive rule in $\boldsymbol{R_i}$ of the form:

---

[7] Note that $\exists Y[p(Y) \vee q(Y)]$ *is equivalent to* $\exists Y\, p(Y) \vee \exists Z\, q(Z)$.

$$antecedent(X) \wedge other\text{-}antecedents(X) \rightarrow consequents(X, Y)$$

where *antecedent(X)* is a conjunction of conditions and event atoms such that there are no conditions or event atoms in *other-antecedents(X)* constrained to be earlier or at the same time as those in *antecedent(X)*,[8] the operational semantics finds all ground instances *antecedent(x)* of *antecedent(X)* that are true in $\boldsymbol{e_i}^* \cup \boldsymbol{S_i}^*$ viewed as a Herbrand model.[9] For each such instance, it generates the corresponding "resolvent":

$$other\text{-}antecedents(x) \rightarrow consequents(x, Y)$$

instantiating all the time variables *Times* that occur in *antecedent(X)* to the current time t_i, simplifying any temporal constraints, and adding the simplified resolvent as a new reactive rule to $\boldsymbol{R_i}$. Simplification consists in deleting any inequalities that are true in $\boldsymbol{Aux}$. $\boldsymbol{R_{i+1}}$ is the resulting expanded set of reactive rules.

If after simplification, *other-antecedents(x)* is an empty conjunction (equivalent to true), then the goal state $\boldsymbol{G_i}$ is updated by adding the simplified *consequents(x, Y)* as a new achievement goal, starting a new thread.

Step 2. Solve Goals.

2.1. Decide. The operational semantics deletes from $\boldsymbol{G_i}$ any disjuncts that are now too late, because they contain conditions whose times are constrained to be earlier than t_i or actions whose times are constrained to be earlier or identical to t_i. If, as a result, all the disjuncts in a thread are deleted, then the thread is false, and the operational semantics terminates in failure.

Otherwise, the operational semantics chooses a set $\boldsymbol{D}$ of disjuncts for evaluation/execution from one or more threads in $\boldsymbol{G_i}$. Choosing disjuncts from different threads amounts to solving several goals concurrently. Choosing disjuncts from the same thread explores different ways of solving the same goal simultaneously.

Step 2.2. Think. For every disjunct in $\boldsymbol{D}$, the operational semantics chooses a form[10]:

$$conditions(Y) \wedge other\text{-}consequents(Y)$$

where *conditions(Y)* is a conjunction of conditions such that there are no conditions or events in *other-consequents(Y)* that are constrained to be earlier or at the same time as those in *conditions(Y)*.

The chosen *conditions* may be empty (equivalent to true), in which case the operational semantics continues with step 3. Otherwise, it queries the current state $\boldsymbol{S_i}$ to determine *some* instance *conditions(y)* of *conditions(Y)* that is true in $\boldsymbol{S_i}^*$. It then generates the corresponding "resolvent":

---

[8] This phrasing allows for the possibility that the *antecedents* of the rule are partially ordered. In such cases, it may be possible to parse a single rule into this form in different ways.

[9] Note that *x* instantiates only those variables in *X* that are in *antecedent(X)*. Any variables in *X* that are not in *antecedent(X)* remain as variables in *x*.

[10] Because all of the universally quantified variables have been instantiated in step 1 to variable-free terms, neither they nor their instances are displayed in this notation.

other-consequents(y)

instantiating the time variables *Times'* in *conditions(Y)* to the current time t_i, simplifying any temporal constraints, and updating both $\boldsymbol{D}$ and $\boldsymbol{G_i}$ by adding the simplified resolvent to the thread in $\boldsymbol{D}$ and $\boldsymbol{G_i}$ containing *conditions(Y)* $\wedge$ *other-consequents(Y).*[11]

Step 3. Act. For every disjunct in $\boldsymbol{D}$ the operational semantics chooses a form:

actions(Z) $\wedge$ *further-consequents(Z)*

where *actions(Z)* is a conjunction of action atoms such that there are no conditions or actions in *further-consequents(Y)* that are constrained to be earlier or at the same time as those in *actions(Z).*

The chosen *actions* may be empty (equivalent to true), in which case the operational semantics continues with step 4. Otherwise, it attempts to execute *actions(Z).* For each such disjunct, if one of the actions in *actions(Z)* fails, then the conjunction of actions fails, and therefore all the actions in *actions(Z)* fail. For all the actions that succeed, it generates the corresponding "resolvent"[12]:

further-consequents(z)

instantiating the time variables *Times''* that occur in *actions(Z)* to the new current time t_{i+1}, simplifying any temporal constraints in *further-consequents(z),* and updating $\boldsymbol{G_i}$ by adding the simplified resolvent to the thread containing *actions(Z)* $\wedge$ *further-consequents(Z).*

If a new disjunct is empty (equivalent to true), the operational semantics deletes the thread containing the disjunct (because the thread is then also equivalent to true).

Step 4. The cycle ends. The current goal state $\boldsymbol{G_i}$ becomes the next goal state $\boldsymbol{G_{i+1}}$, and any successfully executed actions are added to the set of events $\boldsymbol{e_{i+1}}$.

6 Soundness and incompleteness

The soundness of the operational semantics of KELPS can be shown by means of a similar argument to the argument for LPS [17]. The incompleteness of KELPS can also be shown by means of similar counterexamples. In particular, the operational semantics of neither LPS nor KELPS can:

1. preventively make a reactive rule true by making its *antecedents* false, or

[11] Notice that *conditions(Y)* $\wedge$ *other-consequents(Y)* is retained in $\boldsymbol{G_i}$, because *conditions(Y)* can be queried again in the future, as long as no time in *Times'* becomes constrained to be in the past. Moreover it may be possible to try the same disjunct again with a choice of different *conditions(Y).* Similarly, *actions(Z)* $\wedge$ *further-consequents(Z)* is retained in $\boldsymbol{G_i}$ in step 2.3.

[12] The instance *z* of the variables in *Z* can be regarded as a kind of a feedback from the environment.

2. proactively make a reactive rule true by making its *consequents* true before its *antecedents* become true.

These two kinds of incompleteness are tolerable for a simple-minded agent, which can compensate for them by the use of explicit, additional reactive rules, to cater independently for these cases. But they are undesirable for a genuinely intelligent agent.

Examples of these two kinds of incompleteness include:

1. *attacks(X, you, T_1)* $\wedge \neg$ *prepared-for-attack(you, T_1)* $\rightarrow$ *surrender(you, T_2)* $\wedge\ T_1 < T_2 \leq T_1 + \delta$
   KELPS cannot make the rule true by performing actions to make *prepared-for-attack(you, T)* true and so $\neg$ *prepared-for-attack(you, T)* false.

2. *enter-bus(T_1)* $\rightarrow$ *have-ticket(T_2)* $\wedge\ T_1 < T_2 \leq T_1 + \varepsilon$
   KELPS cannot make the rule true by performing actions to make *have-ticket(T_2)* true before *enter-bus(T_1)*.

7 KELPS + LP

KELPS makes the simplifying assumption that auxiliary predicates are defined by possibly infinitely many facts ***Aux***. This has the advantage that it highlights the primary role of reactive rules. But it ignores the need to define auxiliary predicates in a structured, reusable and computationally feasible manner. This need can be filled very simply by logic programs, whose minimal model and perfect model semantics are a natural extension of the model-theoretic semantics of KELPS.

In addition to their use for defining auxiliary predicates, logic programs can be used to define intensional predicates, like view definitions in relational databases. They can also be used to name complex events and actions and to define them both recursively and in terms of other events and actions.

For example, whenever there is an emergency of any kind, we may want to isolate and monitor the area. This can be represented by the reactive rule:

emergency(Area, T_0) $\rightarrow$ *isolate(Area, T_1, T_2)* $\wedge\ T_0 < T_1 < T_2$

and logic program:

emergency(Area, T) $\leftarrow$ *fire(Area, T)* $\vee$ *flood(Area, T)* $\vee$ *noxious-fumes(Area, T)*
isolate(Area, T1, T4) $\leftarrow$ *close-windows(Area, T1, T2)* $\wedge$ *close-doors(Area, T2, T3)* $\wedge$ *lock-doors(Area, T3, T4)* $\wedge\ T1 < T2 < T3 < T4 \wedge$ *focus-camera(Area, T5)* $\wedge\ T1 < T5 \leq T4$

Consider the simplest case, in which the auxiliary predicates are defined by a set of Horn clauses ***LP***. This means that

$$\boldsymbol{M = Aux} \cup \boldsymbol{S_0}^* \cup \boldsymbol{e_1}^* \cup \boldsymbol{S_1}^* \cup \boldsymbol{e_2}^* \cup \ \ldots \ \ \cup \boldsymbol{e_i}^* \cup \boldsymbol{S_i}^* \cup \boldsymbol{e_{i+1}}^* \ \ldots..$$

is the unique minimal model of the set of Horn clauses [3]:

$$LP \cup S_0^* \cup e_1^* \cup S_1^* \cup e_2^* \cup \ \ldots \ \cup e_i^* \cup S_i^* \cup e_{i+1}^* \ldots..$$

The uniqueness of the intended model continues to hold if Horn clause are generalised to locally stratified logic programs [21]. It also continues to hold if negative literals in logic programs are appropriately generalised to formulas of FOL.

As in the case of KELPS, different operational semantics are possible for KELPS + LP. In the case of macro-actions in the consequents of reactive rules, the natural execution method is backward reasoning. However, in the case of macro-events in the antecedents of reactive rules, the natural execution method is a form of forward reasoning triggered by the observation of events. This kind of forward reasoning is similar to integrity checking methods for deductive databases and to the Rete algorithm [5] for production systems. It can be implemented, for example, by means of resolution in the connection graph proof procedure [13].

Earlier papers about LPS [16, 17] contain planning clauses and use intentional predicate definitions to generate future states. The focus in this paper is on defining a version of LPS that is closer to a practical programming language, rather than to a problem description and problem-solving language.

8 MALPS – Multi-Agent LPS

In LPS/KELPS, an agent interacts with a global state, observing and performing updates. This global state can also serve as a communication and coordination medium for a community of agents, like the blackboard model in AI [10], tuple spaces in Linda [2], and conventional, multi-user database management systems.

MALPS (Multi-Agent LPS) [18] exploits this potential of the global state to serve as a coordination medium. Different agents can have different goals, represented by different reactive rules, different beliefs, represented by different logic programs, and different capabilities, represented by different atomic actions.

The extension of LPS/KELPS to MALPS requires no change to the model-theoretic semantics: An action performed by one agent becomes an external event for other agents. Whereas in LPS/KELPS, only a single agent tries to make its own goals true, in MALPS all the agents try to make their goals true.

The use of a global state as a communication and coordination medium contrasts with the use of message-passing in many other computing paradigms, in which the use of global states and global variables is considered an unsafe practice with a problematic semantics. In KELPS, global states not only provide the basis for its operational semantics, but they also underpin its model-theoretic semantics.

The simplest way to implement MALPS is to synchronise updates, so that all of the agent cycles start and end at the same time. This is similar to the simplifying assumption in KELPS that time stands still for the duration of a cycle. This simplifying assumption ensures that, if the only external events are the actions of other agents, then if an agent believes that a condition is true in a given state, then it really is true because no other agent can update the state during the cycle.

9 Conclusions

We have presented a reactive kernel KELPS of LPS that deals with complex events, generalized FOL conditions and temporal constraints.

As we have already mentioned, KELPS is similar to MetateM, with their common focus on generating a model to make a collection of reactive rules true. However, MetateM differs from KELPS by employing modal logic syntax with a possible world semantics, and by using frame axioms to implement change of state.

LPS is also similar to $\mathcal{TR}$ Logic, with their common use of a destructively updated state, and their common view of complex actions as sequences of conditions that are solved by queries and atomic actions that are perform by updates. Indeed, the fact that conditions in LPS can be FOL formulas is largely inspired by $\mathcal{TR}$ Logic.

$\mathcal{TR}$ Logic employs a non-modal syntax, but with a possible world semantics, in which truth values are determined relative to paths in a collection of possible worlds. This semantics is natural for transactions. But it seems cumbersome for reactive rules, whose truth value then depends on whether, for every path over which the *antecedents* are true, there exists a later path over which the *consequents* are true.

MetateM and Transaction Logic are the two computational frameworks that are closest to LPS/KELPS, but they are only the tip of an iceberg. Numerous other attempts have been made to develop a model-theoretic semantics for state transition systems. As far as we can tell, none of them view computation as generating a classical FOL model of a program.

The model-theoretic semantics of FOL, upon which the semantics of LPS is based, is the simplest model-theoretic semantics possible, because all other model-theoretic semantics, including the possible worlds semantics, reduce to it. The use of classical FOL semantics is possible for LPS, because the internal syntax of LPS includes explicit representations of times and events, making it possible to generate single models that include the entire sequence of states and events. The representation of time in the internal syntax can be partially hidden in an external syntax, but can be brought to the surface when necessary, for example to refer to temporal constraints on times and durations. This ability to represent and reason about temporal constraints is essential for practical applications, including those that involve real time and scheduling.

Those frameworks, other than transaction logic, that most obviously provide a model-theoretic semantics for state-transition systems, such as the situation calculus and Metatem, employ frame axioms in the operational semantics. We believe that the ability to update states destructively, without the use of frame axioms, is another important feature of LPS, which is essential for most practical applications. However, the importance of avoiding the use of frame axioms does not seem to be generally appreciated, making it difficult in many cases to compare LPS with other systems. It seems to us that this lack of appreciation is due to a confusion between the representational and computational aspects of the problem.

The *representation aspect of the frame problem* is how to represent formally the property that if a fluent holds in a state, then it continues to hold in future states unless and until it is terminated by an event. Arguably, this aspect of the frame problem has been solved adequately by the use of frame axioms formulated in various non-monotonic logics, including logic programming.

However, it is easy to see that the *computational aspect of the frame problem* of efficiently determining whether a fluent actually holds in a given state cannot be solved in the general case by reasoning with frame axioms. It simply is not computationally feasible either to reason forwards with frame axioms, duplicating facts that hold from one state to the next, or to reason backwards, to determine whether a fact holds in a state by determining whether it held in previous states. The solution of the computational problem requires the use of destructive updates.

LPS/KELPS reconciles the use of destructive updates in the operational semantics with the model-theoretic semantics by making the frame axiom(s) emergent properties, which hold without the need to reason with them operationally.

10 Future work

The operational semantics sketched in this paper is very abstract, and capable of many refinements and optimizations. We have implemented a prototype of LPS that includes some of these improvements, making it closer to a conventional programming language. For example, the implementation uses a Prolog-like depth-first search to choose a singleton set of disjuncts ***D***. Some obvious additional improvements include the use of a constraint solver for handling temporal constraints and the use of a UML-like graphical external syntax.

LPS has its origins in AI knowledge representation and reasoning languages, but has developed into a framework that overlaps with many other areas of computing, including, most notably, database systems, as well as coordination languages that use a shared global state for parallel, concurrent and distributed computing. It is a major challenge to investigate the extent to which LPS might contribute to these areas.

Acknowledgements

We are grateful Harold Boley for helpful comments on an earlier draft of this paper and to Imperial College for EPSRC Pathways to Impact funding, which is supporting the implementation of LPS.

References

1. Bonner and M. Kifer.: Transaction logic programming. In Warren D. S., (ed.), *Logic Programming: Proc. of the 10th International Conf.*, 257-279 (1993)
2. Carriero, N. and Gelernter, D.: Linda in Context. Communications of the ACM. Volume 32 Issue 4, (1989)
3. van Emden, M. and Kowalski, R.: The Semantics of Predicate Logic as a Programming Language, in JACM, Vol. 23, No. 4, 733-742 (1976)
4. Fisher, M.: A Survey of Concurrent METATEM - The Language and its Applications. Lecture notes in computer science, 827, Springer Verlag 480-505 (1994)
5. Forgy, C.L., Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence, volume 19, Issue 1, 17-37 (1982)
6. Fung, T.H. and Kowalski, R. : The IFF Proof Procedure for Abductive Logic Programming. J. of Logic Programming (1997)

7. Harel, D.: Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Programming* 8 231-274 (1987)
8. Harel, D.: Statecharts in the Making: A Personal Account" Proc. *3rd ACM SIGPLAN History of Programming Languages Conference* (HOPL III), (2007)
9. Hausmann, S., Scherr, M., Bry, F.: Complex Actions for Event Processing, Research Report, Institute for Informatics, University of Munich (2012)
10. Hayes-Roth, B.: A blackboard architecture for control, *Artificial Intelligence*, Volume 26, Issue 3, 251-321 (1985)
11. Kakas,T., Kowalski, R., Toni, F.:The Role of Logic Programming in Abduction, *Handbook of Logic in Artificial Intelligence and Programming 5*, Oxford University Press, 235-324 (1998)
12. Knuth, D. E.: The Art of Computer Programming. Vol. 1: Fundamental Algorithms. Addison-Wesley, 1973.
13. Kowalski, R.: *Computational Logic and Human Thinking: How to be Artificially Intelligent,* Cambridge University Press. (2011)
14. Kowalski, R. and Sadri, F.: From Logic Programming Towards Multi-agent Systems, *Annals of Mathematics and Artificial Intelligence*, Volume 25, 391-419 (1999)
15. Kowalski, R. and Sadri, F.: Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents. In *Proceedings of The Third International Conference on Web Reasoning and Rule Systems*, Chantilly, Virginia, USA (2009)
16. Kowalski, R. and Sadri, F.: An Agent Language with Destructive Assignment and Model-Theoretic Semantics, In Dix J., Leite J., Governatori G., Jamroga W. (eds.), *Proc. of the 11th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA*), 200-218 (2010)
17. Kowalski, R. and Sadri, F.: Abductive Logic Programming Agents with Destructive Databases, *Annals of Mathematics and Artificial Intelligence*, Volume 62, Issue 1, 129-158 (2011)
18. Kowalski, R. and Sadri, F.: Programming with logic without logic programming, Department of Computing, Imperial College London Report 2012
19. Kowalski, R., Sergot, M.: A Logic-based Calculus of Events. In: *New Generation Computing*, Vol. 4, No.1, 67--95 (1986).
20. McCarthy, J. and Hayes, P.: Some Philosophical Problems from the Standpoint of Artificial Intelligence, Machine Intelligence 4, Edinburgh University Press. 463-502 (1969)
21. Przymusinski, T.: On the declarative semantics of stratified deductive databases and logic programs. *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann, J. Minker (Ed.) 193 – 216 (1987)
22. Reisig, W.: The Expressive Power of Abstract-State Machines. Computing and Informatics, Vol. 22, 209–219 (2003)