A Guided Tour through some Extensions of the Event Calculus[*]

Iliano Cervesato[†], Massimo Franceschet[‡], and Angelo Montanari[‡]

[†] Department of Computer Science
Stanford University
Stanford, CA 94305-9045
iliano@cs.stanford.edu

[‡] Dipartimento di Matematica e Informatica
Università di Udine
Via delle Scienze, 206 – 33100 Udine, Italy
francesc@dimi.uniud.it; montana@dimi.uniud.it

Abstract

Kowalski and Sergot's Event Calculus (*EC*) is a simple temporal formalism that, given a set of event occurrences, derives the maximal validity intervals (*MVIs*) over which properties initiated or terminated by these events hold. In this paper, we conduct a systematic analysis of *EC* by which we gain a better understanding of this formalism and determine ways of augmenting its expressive power. The keystone of this endeavor is the definition of an extendible formal specification of its functionalities. This formalization has the effects of casting *MVIs* determination as a model checking problem, of setting the ground for studying and comparing the expressiveness and complexity of various extensions of *EC*, and of establishing a semantic reference against which to verify the soundness and completeness of implementations.

We extend the range of queries accepted by *EC*, which is limited to boolean combinations of *MVI* verification or computation requests, to support arbitrary quantification over events and modal queries. We also admit specifications based on preconditions. We demonstrate the added expressive power by encoding a number of diagnosis problems. Moreover, we provide a systematic comparison of the expressiveness and complexity of the various extended event calculi against each other. Finally, we propose a declarative encoding of these enriched event calculi in the logic programming language λ*Prolog* and prove the soundness and completeness of the resulting logic programs.

1 Introduction

The *Event Calculus*, abbreviated *EC* [KS86] is a simple temporal formalism designed to model and reason about scenarios characterized by a set of *events*, whose occurrences have the effect of starting or terminating the validity of determined properties. Events can be temporally qualified in several ways. We consider the case where the time at which they happen is unknown and information about the relative order of their occurrence can be missing [DB88]. Given a (possibly incomplete) description of when these events take place and of the properties they affect, *EC* is able to determine the *maximal validity intervals*, or *MVIs*, over which a property holds uninterruptedly. In practice, since this formalism is usually implemented as a logic program, *EC* can also be used to check the truth of *MVIs* and process boolean combinations of *MVI* verification or computation requests. The range of queries that can be expressed in this way is however too limited for modeling realistic situations.

Several extensions of the basic *EC* have been designed in order to accommodate constructs intended to enhance its expressiveness. In particular, the addition of modal capabilities to *EC* has

[*]The first author was supported by ONR grant N00014-97-1-0505, Multidisciplinary University Research Initiative *Semantic Consistency in Information Exchange*.

been addressed in [CMP93, CMP94, DMB92, Esh88, Sha89]; primitives for dealing with continuous change, discrete processes, and concurrent actions have been proposed in [Eva90, MMCR92, Sha90]; preconditions have been incorporated in different variants of basic *EC* [CFM97b]. However, a uniform framework that allows formally defining and contrasting the expressiveness and complexity of the various extensions to *EC* is still lacking.

In this paper, we unify some of this previous work and carry a systematic analysis of *EC*. The novel understanding of this formalism that emerges from this investigation reveals elegant ways of augmenting its expressive power. The keystone of this endeavor is the definition of an extendible formal specification of functionalities of *EC*. This formalization has the effects of casting *MVIs* derivation in *EC* as a model checking problem, of setting the ground for studying and comparing the expressiveness and complexity of various extensions to *EC*, and of establishing a semantic reference against which to verify the soundness and completeness of implementations. We focus on the integration of boolean connectives, event quantifiers, modalities, and preconditions into *EC*. We first study the properties of extensions that separately incorporate each individual feature; then, we consider the effects of mixing them together.

This paper is organized as follows. In Section 2, we introduce the specification formalism and use it to describe the basic functionalities of *EC*. In Section 3, we formally define a number of extensions of *EC* of increasing expressive power in a uniform way. Section 4 is devoted to exemplifying how the resulting event calculi can adequately model certain diagnosis problems. In Section 5, we thoroughly analyze and contrast the complexity of model checking in the proposed calculi. In Section 6, we devise suitable approximate procedures for those event calculi in which model checking is an intractable problem. In Section 7, we briefly introduce the logic programming language *λProlog*, use it to implement our various event calculi, and prove the soundness and completeness of the resulting logic programs. We outline directions of future work in Section 8.

2 The Basic Event Calculus

The *Event Calculus* (*EC*) [KS86] aims at modeling situations that consist of a set of events, whose occurrences over time have the effect of initiating or terminating the validity of properties, some of which may be mutually exclusive. **[Example from Kowalski]** We formalize the time-independent component of a situation by means of the notion of *EC-structure*, which is defined as follows.

Definition 2.1 (*EC-structure*)

An Event Calculus structure (abbreviated EC-structure*) is a quintuple* $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *such that:*

- $\mathbf{E} = \{e_1, \ldots, e_n\}$ *and* $\mathbf{P} = \{p_1, \ldots, p_m\}$ *are finite sets of* events *and* properties*, respectively.*
- $[\cdot\rangle : \mathbf{P} \rightarrow \mathbf{2}^{\mathbf{E}}$ *and* $\langle\cdot] : \mathbf{P} \rightarrow \mathbf{2}^{\mathbf{E}}$ *are the* initiating *and* terminating maps *of* $\mathcal{H}$*. For every property* $p \in \mathbf{P}$*,* $[p\rangle$ *and* $\langle p]$ *represent the set of events that initiate and terminate* p*, respectively.*
- $]\cdot,\cdot[\subseteq \mathbf{P} \times \mathbf{P}$ *is an irreflexive and symmetric relation, called the* exclusivity relation*, that models exclusivity among properties.* □

The time-dependent component of an *EC* problem is formalized by providing a (strict) *partial order*, i.e. an irreflexive and transitive relation, over the set of event occurrences. We write $W_{\mathcal{H}}$ for the set of all partial orders on the set of events $\mathbf{E}$ of an *EC*-structure $\mathcal{H}$ and use the letter w to denote individual orderings, or *knowledge states*, in $W_{\mathcal{H}}$. The set $W_{\mathcal{H}}$ of all knowledge states naturally becomes a reflexive ordered set when considered together with the usual subset relation $\subseteq$, which is indeed reflexive, transitive, and antisymmetric. Given $w \in W_{\mathcal{H}}$, we will sometimes call a pair of events

$(e_1, e_2) \in w$, with $e_1 \neq e_2$, an *interval.* We will often indicate the condition $(e_1, e_2) \in w$ as $e_1 <_w e_2$, and write $e_1 \leq_w e_2$ for $e_1 <_w e_2 \vee e_1 = e_2$. For any $w_1, w_2 \in W_{\mathcal{H}}$, we denote by $w_1 \uparrow w_2$ the transitive closure of the union of w_1 and w_2, that is, $w_1 \uparrow w_2 = (w_1 \cup w_2)^+$. Note that $w_1 \uparrow w_2 \notin W_{\mathcal{H}}$ if w_1 and w_2 contain symmetric intervals. Finally, we will often work with *extensions* of an ordering w, defined as those elements of $W_{\mathcal{H}}$ which contain w as a subset. We define a *completion* or *final extension* of w as any extension of w which is a total order.

Given an *EC*-structure $\mathcal{H}$ and a knowledge state w, *EC* permits inferring the *maximal validity intervals* (*MVIs*) over which a property p holds uninterruptedly. We represent an *MVI* for p as $p(e_i, e_t)$, where e_i and e_t are the events that respectively initiate and terminate the interval over which p maximally holds. For any given *EC*-structure $\mathcal{H}$, we adopt as the *query language* of *EC*, denoted by $\mathcal{L}_{\mathcal{H}}(\text{EC})$, the set of *atomic formulas* of the form $p(e_1, e_2)$, for all properties p and events e_1 and e_2 in $\mathcal{H}$. The task performed by *EC* reduces to deciding which atomic formulas are *MVI*s and which are not, with respect to the current partial order of events. This is a model checking problem.

In order for $p(e_1, e_2)$ to be an *MVI* relative to the knowledge state w, (e_1, e_2) must be an interval in w, i.e. $e_1 <_w e_2$. Moreover, e_1 and e_2 must witness the validity of the property p at the ends of this interval by initiating and terminating p, respectively. These requirements are enforced by conditions *i*, *ii* and *iii*, respectively, in the definition of valuation given below. The maximality requirement is caught by the negation of the meta-predicate $broken(p, e_1, e_2, w)$ in condition *iv*, which expresses the fact that the truth of an *MVI* must not be *broken* by any interrupting event. Any event e which is known to have happened between e_1 and e_2 in w and that initiates or terminates a property that is either p itself or a property exclusive with p interrupts the truth of $p(e_1, e_2)$. These conditions are formalized as follows.

Definition 2.2 (*Intended model of EC*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be a EC-structure and* w *be a knowledge state in* $W_{\mathcal{H}}$*. The* intended *EC*-model *of* $\mathcal{H}$ *is the propositional valuation* $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \rightarrow 2^{\mathcal{L}_{\mathcal{H}}(\text{EC})}$*, where* $\upsilon_{\mathcal{H}}$ *is defined in such a way that* $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w)$ *if and only if*

i. $e_1 <_w e_2$*;*

ii. $e_1 \in [p\rangle$*;*

iii. $e_2 \in \langle p]$*;*

iv. $broken(p, e_1, e_2, w)$ *does not hold, where* $broken(p, e_1, e_2, w)$ *abbreviates*

> there exists an event $e \in \mathbf{E}$ such that $e_1 <_w e$, $e <_w e_2$ and there exists a property $q \in \mathbf{P}$ such that $e \in [q\rangle$ or $e \in \langle q]$, and either $]p, q[$ or $p = q$. □

This definition adopts the *strong interpretation* of the initiate and terminate relations: given a pair of events e_i and e_t, with e_i occurring before e_t, that respectively initiate and terminate a property p, we conclude that p does not hold over (e_i, e_t) if an event e which initiates or terminates p, or a property incompatible with p, occurs during this interval, that is, (e_i, e_t) is a candidate *MVI* for p, but e forces us to reject it. The strong interpretation is needed when the occurrence of events alter the properties (e.g. turning a switch on and off changes the light in a room). An alternative interpretation of the initiate and terminate relations, called *weak interpretation*, is also possible: a property p is initiated by an initiating event unless it does not already hold and it has not yet terminated (and dually for terminating events). This is useful for events whose occurrence do not influence the validity of a property (e.g. measuring the temperature of a patient does not change it). Further details about the strong/weak distinction can be found in [CM99].

[Formalization of the example]

3 Increasing the Expressiveness of the Basic Event Calculus

The basic Event Calculus has a simple and intuitive logical structure, but its expressive power is too limited to model interesting problems. To overcome these limitations, it has been extended in several directions. In this section, we illustrate the increase in expressive power that can be obtained by adding boolean connectives, quantifiers, modal operators, and preconditions to *EC*. We first consider the extension of *EC* with each of these features taken in isolation, and then we show how they can be combined together. Later on, in Section 5, we will thoroughly analyze the expressiveness and complexity of the resulting event calculi.

3.1 Boolean Connectives

The addition of the logical connectives $\wedge$, $\vee$, and $\neg$ to the basic *EC* makes it possible to check the truth of boolean combinations of *MVI*s. [**Frasetta che estende l'esempio**] Traditional implementations of *EC* exploit the primitive operations of logic programming languages to this effect. The extension of the semantics of *EC* is rather straightforward. We indicate the resulting calculus with *CEC*. Its query language, denoted by $\mathcal{L}_{\mathcal{H}}(\text{CEC})$, is defined as follows.

Definition 3.1 (*CEC-language*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure. The query language* $\mathcal{L}_{\mathcal{H}}(\text{CEC})$ *is the set of formulas generated by the following grammar:*

$$\varphi \ ::= p(e_1, e_2) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \qquad \Box$$

In the sequel, we will also make use of implication, where $\varphi_1 \rightarrow \varphi_2$ is classically defined as $\neg\varphi_1 \vee \varphi_2$.

The definition of the intended model of *EC* can be easily generalized to deal with boolean connectives.

Definition 3.2 (*Intended model of CEC*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure and* w *be a knowledge state in* $W_{\mathcal{H}}$. *The notion of propositional valuation* $\upsilon_{\mathcal{H}}$ *is given as in Definition 2.2. Given* $\varphi \in \mathcal{L}_{\mathcal{H}}(\text{CEC})$, *the truth of* φ *with respect to the intended model of CEC is defined as follows:*

$$\begin{array}{lll}
\mathcal{I}_{\mathcal{H}}; w \models p(e_1, e_2) & \textit{iff} & p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w); \\
\mathcal{I}_{\mathcal{H}}; w \models \neg\varphi & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \not\models \varphi; \\
\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \ \textit{and}\ \mathcal{I}_{\mathcal{H}}; w \models \varphi_2; \\
\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \vee \varphi_2 & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \ \textit{or}\ \mathcal{I}_{\mathcal{H}}; w \models \varphi_2. \qquad \Box
\end{array}$$

3.2 Quantifiers

Another fairly natural extension of basic *EC* can be obtained by adding universal and existential quantifiers over events[1]. We call the resulting formalism the *Event Calculus with Quantifiers* (*QEC* for short). As in the case of boolean connectives, a logic programming implementation of *EC* can emulate existential quantification over individual formulas in $\mathcal{L}_{\mathcal{H}}(\text{EC})$ by means of unification, and

[1]In [CFM98c], we proposed the addition of universal and existential quantifiers over both events and properties. However, quantifiers over property do not appear to enhance significantly the expressiveness of *EC* due to the tight relation between properties and events, hard-coded in the initiation and termination maps. Furthermore, while the set of events that have occurred can grow arbitrarily, the set of relevant properties characterizes the considered application domain and thus it is usually fixed once and for all.

moreover, universally quantified formulas in this language always have trivial solutions. We will see, however, that their combination with other operators leads to fairly complex problems.

In order to accommodate quantifiers, we need to extend the query language of *EC*. We assume the existence of infinitely many *event variables*, that we denote x, possibly subscripted, and we write $\bar{e}$ for a syntactic entity that is either an event in $\mathbf{E}$ or an event variable.

Definition 3.3 (*QEC-language*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure. The query language of QEC, denoted by* $\mathcal{L}_{\mathcal{H}}(\mathrm{QEC})$*, is the set of closed formulas generated by the following grammar:*

$$\varphi \ ::= \ p(\bar{e}_1, \bar{e}_2) \ \mid \ \forall x.\, \varphi \ \mid \ \exists x.\, \varphi \qquad \square$$

The notions of free and bound variables are defined as usual and we identify formulas that differ only by the name of their bound variables. We write $[e/x]\varphi$ for the substitution of an event $e \in \mathbf{E}$ for every free occurrence of the event variable x in the formula φ. Notice that this limited form of substitution cannot lead to variable capture.

We now extend the definition of intended model of *EC* from formulas in $\mathcal{L}_{\mathcal{H}}(\mathrm{EC})$ to objects in $\mathcal{L}_{\mathcal{H}}(\mathrm{QEC})$. To this aim, we need to define the notion of validity for the new constructs of *QEC*.

Definition 3.4 (*Intended model of QEC*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure and* w *be a knowledge state in* $W_{\mathcal{H}}$*. The* intended QEC-model *of* $\mathcal{H}$ *and* w *is the classical model* $\mathcal{I}_{\mathcal{H}}$ *built on top of the valuation* $\upsilon_{\mathcal{H}}$*. Given a (closed) formula* $\varphi \in \mathcal{L}_{\mathcal{H}}(\mathrm{QEC})$*, the truth of* φ *at* $\mathcal{I}_{\mathcal{H}}$*, denoted by* $\mathcal{I}_{\mathcal{H}} \models \varphi$*, is inductively defined as follows:*

$\mathcal{I}_{\mathcal{H}} \models p(e_1, e_2)$ *iff* $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w)$;
$\mathcal{I}_{\mathcal{H}} \models \forall x.\, \varphi$ *iff for all* $e \in \mathbf{E}, \mathcal{I}_{\mathcal{H}} \models [e/x]\varphi$;
$\mathcal{I}_{\mathcal{H}} \models \exists x.\, \varphi$ *iff there exists* $e \in \mathbf{E}$ *such that* $\mathcal{I}_{\mathcal{H}} \models [e/x]\varphi$. $\square$

The well-foundedness of this definition derives from the observation that if $\forall x.\, \varphi$ is a closed formula, so is $[e/x]\varphi$, for every event $e \in \mathbf{E}$, and similarly for the formula $\exists x.\, \varphi$. Observe that, if we reject vacuous quantifications, a formula can contain at most two quantifiers.

It is worth noting that a universal quantification over a finite domain can always be expanded as a finite sequence of conjunctions; similarly, an existentially quantified formula is equivalent to the disjunction of all its instances. This fact hints at the possibility of compiling any *QEC* query to a formula that does not mention any quantifier. Observe, however, that this is possible only after an *EC*-structure has been specified. Therefore, quantifiers are not simply syntactic sugar, but an effective extension over the *EC* query language.

3.3 Modalities

As pointed out in [CMP93], when only partial information about which events have occurred and in what order is available, the sets of *MVI*s derived by *EC* bear little relevance, since the acquisition of additional knowledge about the set of events and/or their occurrence times might both dismiss current *MVI*s and validate new *MVI*s. In [CCM95], Cervesato et al. proposed an extension of *EC* with modal operators, called *Modal Event Calculus* (*MEC* for short), whose query language allows enquiring about which *MVIs* will remain valid in every extension of the current knowledge state, and about which intervals might become *MVIs* in some extension of it. Let us call intervals of these two types *necessary MVIs* and *possible MVIs*, respectively, using $\Box$-*MVIs* and $\Diamond$-*MVIs* as abbreviations. Given an *EC*-structure $\mathcal{H}$, the query language $\mathcal{L}(\mathrm{MEC})$ consists of formulas of the form $p(e_1, e_2)$,

$\Box p(e_1, e_2)$, and $\Diamond p(e_1, e_2)$, for every property p and events e_1 and e_2 in $\mathcal{H}$. They denote the property-labeled interval $p(e_1, e_2)$ as a candidate *MVI*, $\Box$-*MVI*, and $\Diamond$-*MVI*, respectively, and the task of *MEC* reduces to verifying this eventuality.

The intended model of *MEC* is given by shifting the focus from the current knowledge state w to all knowledge states which are accessible from it, i.e., to the set $\text{Ext}_{\mathcal{H}}(w)$. Since $\subseteq$ is a reflexive partial order, $(W_{\mathcal{H}}, \subseteq)$ can be naturally viewed as a finite, reflexive, transitive, and antisymmetric modal frame. This frame, together with the straightforward modal extension of the valuation $\upsilon_{\mathcal{H}}$ to an arbitrary knowledge state, provides a modal model for *MEC*.

Definition 3.5 (*Intended model of MEC*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ be an EC-structure and w be a knowledge state in $W_{\mathcal{H}}$. The intended MEC-model *of $\mathcal{H}$ is the modal model $\mathcal{I}_{\mathcal{H}} = (W_{\mathcal{H}}, \subseteq, \upsilon_{\mathcal{H}})$, where the propositional valuation $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \to 2^{\mathcal{L}_{\mathcal{H}}(\text{EC})}$ is given as in Definition 2.2. Given $w \in W_{\mathcal{H}}$ and $\varphi \in \mathcal{L}_{\mathcal{H}}(\text{MEC})$, the truth of φ at w with respect to $\mathcal{I}_{\mathcal{H}}$, denoted by $\mathcal{I}_{\mathcal{H}}; w \models \varphi$, is defined as follows:*

$$
\begin{array}{lll}
\mathcal{I}_{\mathcal{H}}; w \models p(e_1, e_2) & \textit{iff} & p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w); \\
\mathcal{I}_{\mathcal{H}}; w \models \Box p(e_1, e_2) & \textit{iff} & \textit{for every } w' \in \textit{Ext}(w),\ \mathcal{I}_{\mathcal{H}}; w' \models p(e_1, e_2); \\
\mathcal{I}_{\mathcal{H}}; w \models \Diamond p(e_1, e_2) & \textit{iff} & \textit{there exists } w' \in \textit{Ext}(w) \textit{ such that } \mathcal{I}_{\mathcal{H}}; w' \models p(e_1, e_2).
\end{array}
$$
□

In the following, we will drop the subscripts $_{\mathcal{H}}$ whenever this does not lead to ambiguities. Moreover, given a knowledge state w in $W_{\mathcal{H}}$ and a *MEC*-formula φ over $\mathcal{H}$, we write $w \models \varphi$ for $\mathcal{I}_{\mathcal{H}}; w \models \varphi$.

In [CM99], Cervesato and Montanari have shown that, given an *EC*-structure $\mathcal{H}$ and $w \in W_{\mathcal{H}}$, the sets of $\Box$- and $\Diamond$-*MVIs* can be determined by exploiting necessary and sufficient *local conditions* over w, thus avoiding a complete (and expensive) search of the set $\text{Ext}_{\mathcal{H}}(w)$ of all the consistent extensions of w.

Lemma 3.6 (*Local conditions*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ be a EC-structure. For any atomic formula $p(e_1, e_2)$ on $\mathcal{H}$ and any $w \in W_{\mathcal{H}}$,

- $\mathcal{I}_{\mathcal{H}}; w \models \Box p(e_1, e_2)$ *if and only if*
  - *i′.* $e_1 <_w e_2$;
  - *ii′.* $e_1 \in [p\rangle$;
  - *iii′.* $e_2 \in \langle p]$;
  - *iv′.* *necBroken*(p, e_1, e_2, w) *does not hold, where necBroken*(p, e_1, e_2, w) *abbreviates*

    there exists an event $e \in \mathbf{E}$ such that $e \not<_w e_1$, $e \neq e_1$, $e_2 \not<_w e$, $e \neq e_2$, and there exists a property $q \in \mathbf{P}$ such that $e \in [q\rangle$ or $e \in \langle q]$, and either $]p, q[$ or $p = q$.

- $\mathcal{I}_{\mathcal{H}}; w \models \Diamond p(e_1, e_2)$ *if and only if*
  - *i″.* $e_2 \not<_w e_1$;
  - *ii″.* $e_1 \in [p\rangle$;
  - *iii″.* $e_2 \in \langle p]$;
  - *iv″.* *broken*(p, e_1, e_2, w) *does not hold.* ■

3.4 Preconditions

In many application domains, the occurrence of an event is no guaranty that a property is initiated or terminated, because the actual production of the expected effects of the event are tied to the validity of a number of other properties, called *preconditions*, at its occurrence time. In the following, we introduce the *Event Calculus with Preconditions* (*PEC*), which adds preconditions to basic *EC*. Unlike the previous cases, this extension requires a generalization of the notion of *EC*-structure to take into account the contexts within which an event occurs. To model contexts, we replace the notion of *EC*-structure by that of *PEC*-structure, which is defined as follows.

Definition 3.7 (*PEC-structure*)

A structure *for the Event Calculus with Preconditions* (*abbreviated PEC*-structure) *is a quadruple* $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$ *such that:*

- $\mathbf{E} = \{e_1, \ldots, e_n\}$ *and* $\mathbf{P} = \{p_1, \ldots, p_m\}$ *are finite sets of* events *and* properties*, respectively. The subsets of* $\mathbf{P}$ *are called* contexts *and their elements are referred to as* preconditions*.*
- $[\cdot|\cdot\rangle : \mathbf{P} \times \mathbf{2^P} \rightarrow \mathbf{2^E}$ *and* $\langle\cdot|\cdot] : \mathbf{P} \times \mathbf{2^P} \rightarrow \mathbf{2^E}$ *are the* initiating *and* terminating maps *of* $\mathcal{H}$*. For every property* $p \in \mathbf{P}$*,* $[p|C\rangle$ *and* $\langle p|C]$ *represent the set of events that respectively initiate and terminate* p*, whenever all preconditions in* C *hold at their occurrence time.* □

Preconditions can easily emulate the exclusivity relation. Indeed, incompatibility among a pair of properties p and q can be expressed in *PEC* by means of an auxiliary property s_{pq} that acts as a semaphore between p and q: s_{pq} is initially true; every event that starts p or q is equipped with s_{pq} as a precondition, and, besides activating either original property, reset s_{pq}; events that terminate p or q are treated dually. Since exclusivity can be handled in this way, this relation is not included in the definition of *PEC*-structure. Clearly, in the absence of incompatible properties, an *EC* problem can be modeled by a degenerated *PEC*-structure where all contexts are empty.

Given a *PEC*-structure $\mathcal{H}$, the *query language* of *PEC* coincides with the query language of *EC*, that is, it consists of the set of atomic formulas of the form $p(e_1, e_2)$, for all properties p and events e_1 and e_2 in $\mathcal{H}$. The definition of *MVI* differs from the case of *EC* by the way the initiation and termination of a property is checked. Indeed, these relations are now conditional with respect to the validity of a set of preconditions. Bullets *ii* and *iii* below formalize this intuition.

Definition 3.8 (*Intended model of PEC*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$ *be a* PEC*-structure and* w *be a knowledge state in* $W_\mathcal{H}$*. An* intended PEC-model *of* $\mathcal{H}$ *is any propositional valuation* $\upsilon_\mathcal{H} : W_\mathcal{H} \rightarrow 2^{\mathcal{L}_\mathcal{H}(\mathrm{EC})}$ *defined in such a way that* $p(e_1, e_2) \in \upsilon_\mathcal{H}(w)$ *if and only if*

i. $e_1 <_w e_2$;

ii. $pInit(e_1, p, w)$*, where* $pInit(e_1, p, w)$ *iff*
$$\exists C \in \mathbf{2^P}.\ e_1 \in [p|C\rangle \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.$$
$$q(e', e'') \in \upsilon_\mathcal{H}(w) \quad \wedge \quad e' <_w e_1 \quad \wedge \quad e_1 \leq_w e'';$$

iii. $pTerm(e_2, p, w)$*, where* $pTerm(e_2, p, w)$ *iff*
$$\exists C \in \mathbf{2^P}.\ e_2 \in \langle p|C] \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.$$
$$q(e', e'') \in \upsilon_\mathcal{H}(w) \quad \wedge \quad e' <_w e_2 \quad \wedge \quad e_2 \leq_w e'';$$

iv. $pBroken(p, e_1, e_2, w)$ *does not hold, where* $pBroken(p, e_1, e_2, w)$ *iff*
$$\exists e \in \mathbf{E}. \quad e_1 <_w e \quad \wedge \quad e <_w e_2 \quad \wedge \quad (pInit(e, p, w) \vee pTerm(e, p, w)).$$ □

Notice that the endpoints of an interval are not treated symmetrically. This implements our convention according to which a property does not hold at the occurrence time of the event that initiates it, while it must hold at the occurrence time of the event that terminates it.

The meta-predicates *pInit*, *pTerm*, and *pBroken* are mutually recursive in the above definition. In particular, an attempt at computing *MVI*s by simply unfolding their definition is non-terminating

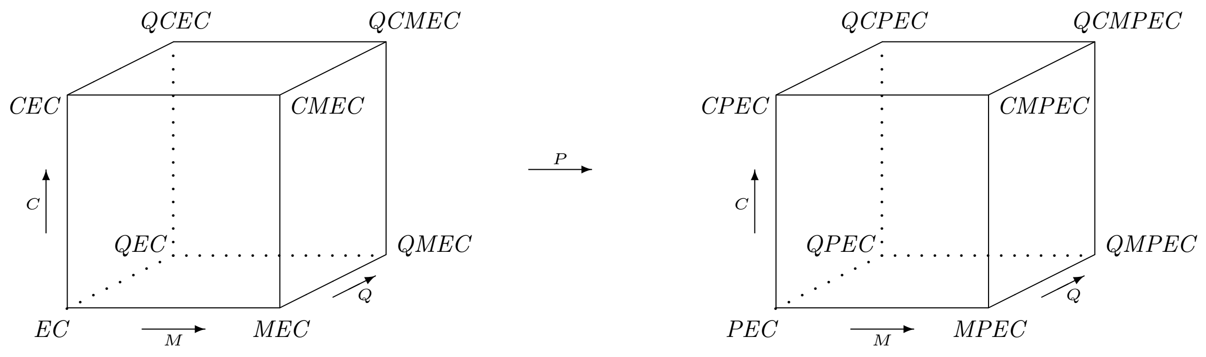

Figure 1: The *EC* Cubes: a Summary of the Proposed Event Calculi

in pathological situations. In general, a *PEC* problem can have zero or more models. However, most *PEC* problems encountered in practice satisfy syntactic conditions ensuring the termination of this procedure and the uniqueness of the model. This is particularly important since it permits the transcription of the above specification as a logic program that is guaranteed to terminate. We need the following definition.

Definition 3.9 (*Dependency Graph*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$ *be a PEC-structure. The* dependency graph *of* $\mathcal{H}$*, denoted by* $G_{\mathcal{H}}$*, consists of one node for each property in* $\mathbf{P}$*, and contains the edge* (q,p) *if and only if the following meta-formula holds* $\exists e \in \mathbf{E}.\ \exists C \in \mathbf{2}^{\mathbf{P}}.\ q \in C \wedge (e \in [p|C\rangle \vee e \in \langle p|C])$. □

In the following, we will restrict our attention to those *PEC*-structures $\mathcal{H}$ such that $G_{\mathcal{H}}$ is acyclic. Under such an assumption, for every property $p \in \mathbf{P}$, the length of the longest path to p in $G_{\mathcal{H}}$ is finite. We denote it as $B_{\mathcal{H}}(p)$. Furthermore, we set $B_{\mathcal{H}} = \max_{p \in \mathbf{P}} B_{\mathcal{H}}(p)$ and write $C_{\mathcal{H}}$ for the cardinality of the largest context in $[\cdot|\cdot\rangle$ or $\langle\cdot|\cdot]$. Finally, we denote with $D_{\mathcal{H}}$ the maximum number of contexts in which an event initiates or terminates a property with respect to the structure $\mathcal{H}$. It is worth noting that the above restriction ensures that the computation of any *MVI* on the basis of Definition 3.8 can never contain more than $B_{\mathcal{H}}$ embedded *MVI* calculations and therefore it always terminates.

3.5 Mixing Extended Functionalities

Only the simplest of problems can be expressed in the extended event calculi described so far. However, many interesting situations can be modeled by combining two or more of the functionalities we have introduced. The modular structure of the given formalization makes it possible to compose the proposed elementary extensions by simply merging their semantic definitions. We will refer to these hybrid calculi by prefixing the string "*EC*" with any subsequence of the letters *Q*, *C*, *M*, and *P*, standing for the inclusion of quantifiers, connectives, modalities, and preconditions, respectively. For example, the "first-order" event calculus, denoted *QCEC*, is obtained by adding boolean connectives and quantifiers to the basic *EC*. Its query language is obtained by merging the productions of the grammars for *CEC* and *QEC*, which then allows an arbitrary interleaving of as many quantifiers and connectives as desired. Similarly, its intended model results from accumulating the semantic clauses of those two calculi.

The inclusion of modalities yields various flavors of a formalism that we have proven to be an instance of the modal logic *K1.1*, a close relative of *S4* whose models are finite reflexive partial orderings (see [CM99] for further details), in the case of *CMEC* [CM99]. Differently from the other elementary extensions, the addition of preconditions to a calculus does not change its query language, however it forces us to modify its underlying structure and intended model. We have taken this distinction into account in Figure 1, where we give a structured representation of our family of calculi. The cube on the left relates all event calculi devoid of preconditions, while the language-wise isomorphic cube on the right shows the corresponding calculi with preconditions. The most expressive language in this hierarchy is *QCMPEC*, which includes boolean connectives, quantifiers, modalities, and preconditions.

Elsewhere, we have investigated meaningful subsets of the event calculi in Figure 1. In [CCM95, CCM96, CFM97a, CM99], we concentrated on propositional modal event calculi, while, in [CFM98a, CFM98b], we studied the interplay of modalities and quantifiers. Preliminary work on modal event calculi with preconditions and first-order event calculus has been reported in [CFM97b] and [CFM98c], respectively. In this paper, we give a comprehensive and systematic analysis of the expressiveness and complexity of the whole family of event calculi. In Section 7, we will provide all calculi with a simple modular implementation in *λProlog*. This implementation will take advantage of some standard logical equivalences of *S4* and few specific to *K1.1*.

Proposition 3.10 (*QCMPEC logical equivalences*)

Let φ, φ_1, and φ_2 be QCMPEC-formulas. For every knowledge state $w \in W$, it holds that

1.	$w \models \Box\neg\varphi$	$\Leftrightarrow$	$w \models \neg\Diamond\varphi$		*2.*	$w \models \Diamond\neg\varphi$	$\Leftrightarrow$	$w \models \neg\Box\varphi$	
3.	$w \models \Box(\varphi_1 \wedge \varphi_2)$	$\Leftrightarrow$	$w \models \Box\varphi_1 \wedge \Box\varphi_2$		*4.*	$w \models \Diamond(\varphi_1 \vee \varphi_2)$	$\Leftrightarrow$	$w \models \Diamond\varphi_1 \vee \Diamond\varphi_2$	
5.	$w \models \Box\Box\varphi$	$\Leftrightarrow$	$w \models \Box\varphi$		*6.*	$w \models \Diamond\Diamond\varphi$	$\Leftrightarrow$	$w \models \Diamond\varphi$	
7.	$w \models \Box\Diamond\Box\varphi$	$\Leftrightarrow$	$w \models \Box\Diamond\varphi$		*8.*	$w \models \Diamond\Box\Diamond\varphi$	$\Leftrightarrow$	$w \models \Diamond\Box\varphi$	
9.	$w \models \Box\forall X\varphi(X)$	$\Leftrightarrow$	$w \models \forall X\Box\varphi(X)$		*10.*	$w \models \Diamond\exists x.\varphi(x)$	$\Leftrightarrow$	$w \models \exists x.\Diamond\varphi(x)$	■

An interesting consequence of Proposition 3.10 is that any *QCMPEC*-formula φ is logically equivalent to a formula of one of the following forms: ψ, $\Box\psi$, $\Diamond\psi$, $\Box\Diamond\psi$, and $\Diamond\Box\psi$, where the outermost operator of ψ is non-modal.

4 Case studies

[Applicazioni in ambito medico — Luca, Peressi, ...] In this section, we take advantage of the increased expressive power of our extensions to the basic Event Calculus to represent and query three situations. The first two make an essential use of preconditions, while the third relies on quantifiers.

4.1 Diagnosing Faulty Hardware

We first focus our attention on the representation and processing of information about fault symptoms that is spread out over periods of time and for which current expert system technology is particularly deficient [Nök91]. Consider the following example, which diagnoses a fault in a computerized numerical control center (CNCC) for a production chain.

> *A possible cause for an undefined position of a tool magazine is a faulty limit switch S. This cause can be ruled out if the status registers IN29 and IN30 of the control system show the following behavior: at the beginning both registers contain the value* 1*. Then IN29 drops to* 0*, followed by IN30. Finally, both return to their original values in the reverse order.*

Figure 2 describes a possible sequence of transitions, for *IN29* and *IN30*, that excludes the eventuality of *S* being faulty. In order to verify this behavior, the contents of the status registers must be monitored over time. Typically, measurements are made at fixed intervals, asynchronously with

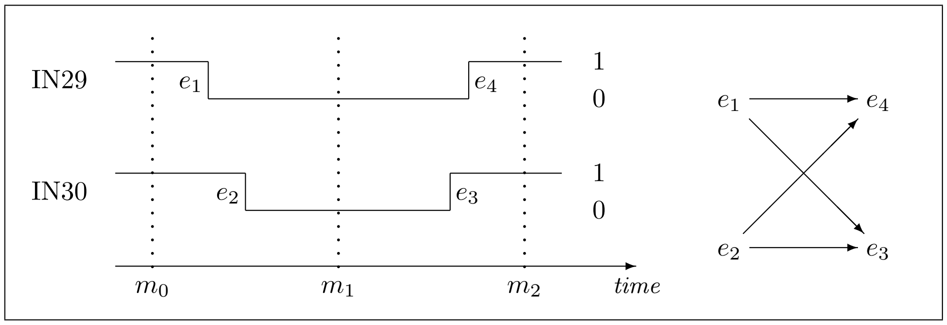

Figure 2: Expected Register Behavior, Measurements and Resulting Event Ordering

respect to the update of status registers. This is primary or routine monitoring. While primary measurements can be taken frequently enough to guarantee that signal transitions are not lost, it is generally impossible to exactly locate the instants at which a register changes its value. Consequently, several transitions may take place between two measurements, making it impossible to recover their relative order. In the case of our example, the situation is depicted in Figure 2 (left): dotted lines indicate measurements. Moreover, we have given names to the individual transitions of state of the different registers. From the values found at measurements m_0 and m_1, we can conclude that both *IN29* and *IN30* were reset during this interval (transitions e_1 and e_2, respectively), but we have no information about their relative ordering. Similarly, measurement m_2 informs us that the registers assumed again the value 1 (transitions e_3 and e_4), but we do not know which was set first. The available ordering information is reported on the right-hand side of Figure 2.

It is conceivable that the system at hand permits finer forms of measurement that would allow resolving the relative order of critical events, but at a substantial cost. This may involve shutting it off and restarting it in a much slower debugging mode, feeding the overall trace to an expensive algorithm, or calling into action a human expert. Either of these actions is too costly to be part of the normal operation of the system, while checking the value of the registers at fixed interval is an acceptable overhead. However, whenever the routine measurement indicates that the system may be going awry, more precise data can be obtained through this second level monitoring. Clearly, suspected faults should be infrequent enough to make it a viable solution.

The situation displayed in Figure 2 is represented by the *PEC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$, whose components are defined as follows:

- $\mathbf{E} = \{e_1, e_2, e_3, e_4\}$
- $\mathbf{P} = \{one29, zero29, one30, zero30\}$
- $\{e_1\} = [zero29|\{\}\rangle$
  $\{e_2\} = [zero30|\{zero29\}\rangle$
  $\{e_3\} = [one30|\{\}\rangle$
  $\{e_4\} = [one29|\{\}\rangle$
- $\{e_1\} = \langle one29|\{\}]$
  $\{e_2\} = \langle one30|\{\}]$
  $\{e_3\} = \langle zero30|\{zero29\}]$
  $\{e_4\} = \langle zero29|\{\}]$

We have represented transitions as events with the same name, and used mnemonic constants for the properties corresponding to the two different values of *IN29* and *IN30*. It is easy to check that the dependency graph for $\mathcal{H}$ does not contain any loop.

It is worth noting that, in general, preconditions do not imply physical sequentiality. As an example, stating that the event e_2 initiates the property $zero30$ only if the property $zero29$ holds expresses the fact that we are only interested in those situations where *IN30* is reset while *IN29* holds the value 0. In such a way, we are able to a priori eliminate a number of incorrect behaviors.

The partial order of transitions, described in Figure 2 (right), is captured by the following (current) knowledge state:

$$o = \{(e_1, e_3), (e_1, e_4), (e_2, e_3), (e_2, e_4)\}.$$

Let us consider the *PEC*-formula:

$$\varphi = zero30(e_2, e_3).$$

In order to verify that the switch S is not faulty, we must ensure that the registers $IN29$ and $IN30$ display the expected behavior in all refinements of the current knowledge state o. With our encoding, this amounts to proving that the *MPEC*-formula $\Box\varphi$ holds in o. If this is the case, the fault is to be excluded. If we want to determine the existence of at least one extension of o where the registers behave correctly, we must verify the satisfiability of the *MPEC*-formula $\Diamond\varphi$ in o. If this is not the case, the fault is certain. Since we have that $o^+ \models \Diamond\varphi$ and $o^+ \not\models \Box\varphi$, the knowledge available in o entitles us to assert that the fault is possible, but not certain. We need to run second-level monitoring to determine the relative order of the unrelated events. Assume that, unlike in the actual situation of Figure 2, e_2 precedes e_1. Let us denote the resulting state by o_1. It holds that $o_1^+ \not\models \Diamond\varphi$, and thus the switch S is certainly faulty. On the other hand, if the actual ordering contains the pairs (e_1, e_2) and (e_3, e_4), calling o_2 the resulting state, we have that $o_2^+ \models \Box\varphi$. In this case the fault can be excluded.

4.2 Diagnosing Metatropic Dwarfism

As a second example, consider the following situation of illnesses taken from the domain of diagnosis of skeletal dysplasias [KW90].

The model of the Metatropic Dwarfism specifies that at birth the thorax is narrow and after the first year of age a mild kyphoscoliosis occurs. If the severity of the kyphoscoliosis is relatively mild then the thorax will continue to be narrow. If the severity of the kyphoscoliosis increases, then there is a period during which the thorax is perceived as relatively normal, but when the kyphoscoliosis is progressive the thorax becomes wide. Metatropic Dwarfism can be excluded if the symptoms do not comply to this model.

Figure 3 schematizes the evolution of a patient to be diagnosed with Metatropic Dwarfism. Both kyphoscoliosis severity and thorax width are continuous attributes, but radiologists are only interested in a finite set of discrete qualitative values (*narrow*, *normal*, and *wide* for the thorax; *mild*, *moderate*, and *progressive* for the scoliosis), and hence only the events which mark the transitions from one qualitative value to the next one are significant. In order to verify this model, the width of the thorax and the severity of the kyphoscoliosis must be checked over time. However, as in the case of measurements of status registers, while the radiological examinations can be done frequently enough to guarantee that qualitative value transitions are not lost, it is generally impossible to exactly locate the instants at which these transitions happen. Consequently, several transitions may take place between two examinations making it impossible to recover their relative order. In the case of our example, the situation is depicted in Figure 3. Exams x_0 and x_1 tell us respectively that at birth the thorax was narrow and that after the first year a mild kyphoscoliosis had developed. We denote with e_0 and e_1 the corresponding events. With exam x_2, we observe that the thorax is now normal and the kyphoscoliosis has become moderate. We write e_3 and e_2 for the corresponding events. We know that they have occurred after e_1, but we have no information about their relative ordering. Finally, exam x_3 informs us that the thorax has successively become wide and the kyphoscoliosis progressive. Let e_5 and e_4 be the corresponding causing events. Again, we know they have happened after e_2 and e_3,

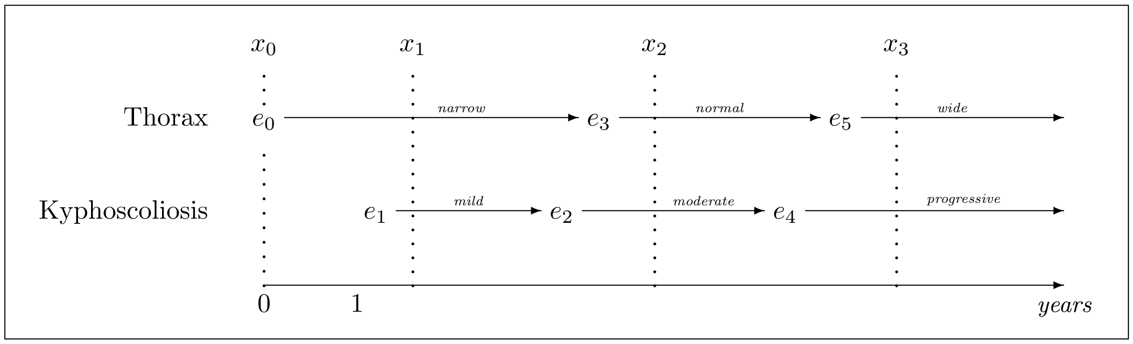

Figure 3: Expected Symptom Evolution for Metatropic Dwarfism

however we are not able to order them. As in the previous example, the exact order in which these transitions happen can be determined by further clinical examinations.

The situation displayed in Figure 3 can be represented by the *PEC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$, whose components are defined as follows:

- $\mathbf{E} = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6\}$
- $\mathbf{P} = \{narrow,\ normal,\ wide,\ mild,\ moderate,\ progressive\}$
- $\{e_0\} = [narrow|\{\}\rangle$
  $\{e_1\} = [mild|\{\}\rangle$
  $\{e_2\} = [moderate|\{\}\rangle$
  $\{e_3\} = [normal|\{moderate\}\rangle$
  $\{e_4\} = [progressive|\{\}\rangle$
  $\{e_5\} = [wide|\{progressive\}\rangle$
- $\{e_2\} = \langle mild|\{\}]$
  $\{e_3\} = \langle narrow|\{\}]$
  $\{e_4\} = \langle moderate|\{\}]$
  $\{e_5\} = \langle normal|\{\}]$
  $\{e_6\} = \langle wide|\{\}] = \langle progressive|\{\}]$

We have added the event e_6 in order to terminate the validity of the properties *wide* and *progressive*; it corresponds to the death of the patient. As in the previous example, our use of preconditions is instrumental to the inferences we want to achieve. Finally, observe that the dependency graph for $\mathcal{H}$ does not contain loops. The partial order of transitions, described in Figure 3, is captured by the following (current) knowledge state:

$$o = \{(e_0, e_1), (e_1, e_2), (e_1, e_3), (e_2, e_4), (e_2, e_5), (e_3, e_4), (e_3, e_5), (e_4, e_6), (e_5, e_6)\}.$$

Consider the *CMPEC*-formula:

$$\varphi \ = \ normal(e_3, e_5) \wedge wide(e_5, e_6).$$

In order to verify that the diagnosis of the dysplasia is certain, we must ensure that the *CMPEC*-formula $\Box\varphi$ is satisfiable in o. If we want to determine if it is possible to diagnose the dysplasia, we must verify the satisfiability of the *CMPEC*-formula $\Diamond\varphi$ in o. Since we have that $o^+ \models \Diamond\varphi$ and $o^+ \not\models \Box\varphi$, the knowledge contained in o entitles us to assert that the diagnosis of the dysplasia is possible, but not certain. Assume that, unlike the actual situation of Figure 3, further examinations extend o with the pair (e_3, e_2). Let us denote the resulting state with o_1. It is easy to prove that

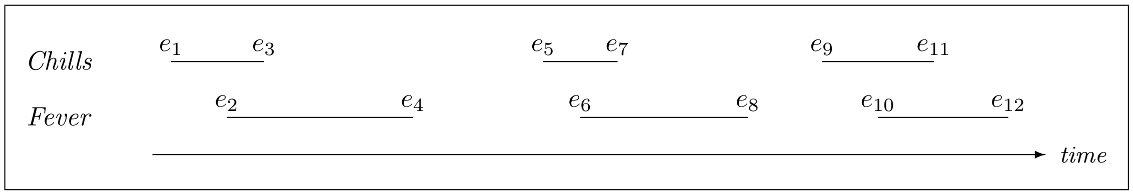

Figure 4: Symptoms of Patient Jones

$o_1^+ \not\models \Diamond\varphi$, and thus that the dysplasia can be excluded. On the other hand, if o is refined with the pairs (e_2, e_3) and (e_4, e_5), and o_2 is the resulting state, we have that $o_2^+ \models \Box\varphi$. In this case, the dysplasia is certain.

4.3 Diagnosing Malaria

We will now consider another example taken from the domain of medical diagnosis that shows how an extension of *EC* with quantifiers and connectives is applied.

We focus our attention on repeated clusters of events whose correlation, if present, can entail conclusions about the state of the system under observation. As an example, consider the following rule of thumb for diagnosing malaria [Sch95]:

> *A malaria attack begins with chills that are followed by high fever. Then the chills stop and some time later the fever goes away as well. Malaria is likely if the patient has repeated episodes of malaria attacks.*

Figure 4 describes the symptoms of a patient, Mr. Jones, who has just returned from a vacation to the Tropics. We have labeled the beginning and the end of chills and fever periods for reference. According to the rule above, Mr. Jones should be diagnosed with malaria. If however he had not had fever in the period between e_6 and e_8 for example, or if e_7 had preceded e_6, then further checks should be made in order to ascertain the kind of ailment he suffers from. Notice that, in this situation, we know when each event has occurred, and therefore their exact relative order.

We will now show how the rule above can be expressed as a *QCEC* query in order to automate the diagnosis of malaria. For the sake of readability, we slightly extend *QCEC* by permitting queries to explicitly test the relative order of two events [CFM98c]. To this end, it suffices to add the production

$$\varphi ::= \bar{e}_1 < \bar{e}_2 \mid \ldots$$

to the grammar defining the language $\mathcal{L}_{\mathcal{H}}(\text{QCEC})$, and the semantic clause

$$\mathcal{I}_{\mathcal{H}} \models e_1 < e_2 \quad \textit{iff} \quad e_1 <_w e_2$$

to the definition of the *QCEC* intended model. Any language in Figure 1 can be enriched to accomodate this construct, called *precedence test*. Observe however that adding it does not change their expressive power as it can be emulated by means of extra events and properties.

The first task is to give a representation to the symptoms. In the case of Mr. Jones, the factual information of his condition is represented by the *EC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ below, which is a direct transliteration of the data in Figure 4.

- $\mathbf{E} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$
- $\mathbf{P} = \{\mathit{chills}, \mathit{fever}\}$
- $[\mathit{chills}\rangle = \{e_1, e_5, e_9\}$
  $[\mathit{fever}\rangle = \{e_2, e_6, e_{10}\}$

- $\langle chills] = \{e_3, e_7, e_{11}\}$
  $\langle fever] = \{e_4, e_8, e_{12}\}$
- $]\cdot,\cdot[= \emptyset$

The events that initiate and terminate the symptoms of Mr. Jones happened in ascending order of their indices. We call w the corresponding ordering.

The decision rule for diagnosing malaria can then be reworded as saying that "*whenever there is an episode of chills, there is a successive period of fever that starts before the chills are over*" (other possible interpretations can easily be rendered in *QCEC*). It can in turn be expressed by the following formula in $\mathcal{L}_{\mathcal{H}}$(QCEC):

$$\varphi \;=\; \forall x_1.\, \forall x_2.\, (chills(x_1, x_2) \rightarrow \exists x_1'.\, \exists x_2'.\, (x_1 < x_1' \;\wedge\; x_1' < x_2 \;\wedge\; fever(x_1', x2')))$$

that makes use of both universal and existential quantifiers over events, of all the connectives of *QCEC* (once implication is expanded) and of the precedence test. It is easy to verify that $\mathcal{I}_{(\mathcal{H},w)} \models \varphi$, while this formula is not valid in models where e_6 or e_8 have been eliminated, or where the relative order of e_6 and e_7 has been reversed, for example.

5 Complexity Analysis

In this section, we systematically analyze the worst-case computational complexity of model checking in the proposed event calculi. The analysis is based on the model-theoretic characterization of the various event calculi provided in Sections 2 and 3. We assume the reader to be familiar with the basics of computational complexity theory [Pap94]. We only remind the definition of *polynomial hierarchy*. The complexity class $\mathbf{P}^{\mathbf{NP}}$ (resp. $\mathbf{NP}^{\mathbf{NP}}$) contains all the problems for which there exists a deterministic (resp. non-deterministic) polynomial time algorithm that makes a number calls to an $\mathbf{NP}$-oracle that is polynomial in the size of the input. The class $\mathbf{coNP}^{\mathbf{NP}}$ is the set of the complements of the problems in $\mathbf{NP}^{\mathbf{NP}}$. We then define the complexity classes Δ_i, Σ_i, and Π_i, for $i \geq 0$, as follows:

$$\left\{\begin{array}{lcl} \Delta_0\mathbf{P} & = & \mathbf{P} \\ \Sigma_0\mathbf{P} & = & \mathbf{P} \\ \Pi_0\mathbf{P} & = & \mathbf{P} \end{array}\right. \qquad \left\{\begin{array}{lcl} \Delta_{i+1}\mathbf{P} & = & \mathbf{P}^{\Sigma_i\mathbf{P}} \\ \Sigma_{i+1}\mathbf{P} & = & \mathbf{NP}^{\Sigma_i\mathbf{P}} \\ \Pi_{i+1}\mathbf{P} & = & \mathbf{coNP}^{\Sigma_i\mathbf{P}} \end{array}\right.$$

The set $\mathbf{PH} = \bigcup_{i\geq 0} \Sigma_i\mathbf{P}$ is called the *cumulative polynomial hierarchy*. It holds that $\mathbf{PH} \subseteq \mathbf{PSPACE}$. **[Check Δ_i and Π_i]**

Given an *EC*-structure $\mathcal{H}$ (or a *PEC*-structure $\mathcal{H}$, if the event calculus under consideration encompasses preconditions), we assume that the set of event occurrences $\mathbf{E}$ can grow arbitrarily, while the set $\mathbf{P}$ of relevant properties characterizes the specific application domain and thus it is fixed once and for all. Thus, we choose the number n of events in $\mathbf{E}$ as the size of $\mathcal{H}$, and consider the number of properties as a constant. Notice that, in the case of event calculi with preconditions, such an assumption allows us to identify upper bounds for the parameters $B_{\mathcal{H}}$, $C_{\mathcal{H}}$, and $D_{\mathcal{H}}$ which do not depend on the number of events in $\mathbf{E}$.

Given a structure $\mathcal{H}$, a knowledge state $w \in W_{\mathcal{H}}$, and a formula φ relative to any of the proposed event calculi, we want to study the complexity of the problem of establishing whether $\mathcal{I}_{\mathcal{H}}; w \models \varphi$ is true or not, which is a model checking problem. We measure the complexity of testing whether $\mathcal{I}_{\mathcal{H}}; w \models \varphi$ holds in terms of the size n of the input database, where n is the number of events in $\mathbf{E}$, and the size k of the input formula (without loss of generality, we can interpret k as the number of logical operators occurring in φ). In the standard terminology, n and k are the parameters that characterize *data* and *query complexity*, respectively.

For each event calculus in Figure 1, we establish whether model checking is a tractable problem or not. In the positive case, we actually provide a polynomial upper bound to the problem by exhibiting a polynomial algorithm that solves it; in the negative case, we identify the complexity classes the intractable problems belong to and investigate the (possibly different) role of the query and data complexity parameters.

The notion of cost we adopt is as follows: we assume that verifying the truth of the propositions $e \in [p\rangle$, $e \in [p|C\rangle$, $e \in \langle p]$, $e \in \langle p|C]$, and $]p, q[$ has constant cost $\mathcal{O}(1)$. We have two choices as far as $e_1 <_w e_2$ is concerned, corresponding to two alternative ways to represent the relation $<_w$:

- Recording $<_w$ as specified by its definition, in particular as a closed transitive relation, has the advantage that checking whether $e_1 <_w e_2$ has constant cost $\mathcal{O}(1)$. The pitfalls of this representation are that it will in general require a lot of space, and that updating it with an edge (e_1, e_2) costs $\mathcal{O}(n^2)$ since the transitive closure $w \uparrow \{(e_1, e_2)\}$ has to be regenerated. This possibility has however been successfully pursued by showing that the graph-theoretic notions of transitive closure and reduction can be exploited to efficiently reason about partially ordered events in *EC* and *MEC* [FM99b].

- As an alternative, we can record an acyclic binary relation o on events, whose transitive closure o^+ is the current state of knowledge w. Then, an update (e_1, e_2) of the current acyclic relation o is implemented in time $\mathcal{O}(1)$ by simply taking the union $o \cup \{(e_1, e_2)\}$. However, verifying the truth of $e_1 <_w e_2$ becomes an accessibility problem in o, which can be solved in $\mathcal{O}(n^2)$ time, where n is the number of event occurrences, as shown in [CMC95, FM99a].

We will report our results for both options. We will perform our calculations for the first, but enclose the figures corresponding to the second in square brackets. This distinction makes sense only for calculi for which model checking can be solved in polynomial time.

Theorem 5.1 (*Polynomial event calculi*)

Model checking in EC*,* CEC*,* QEC*,* MEC*,* PEC*,* QMEC*,* CPEC*, and* QPEC *is polynomial-time bound.*

Proof.

EC: $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$

Let $\mathcal{H}$ be an *EC*-structure, $p(e_1, e_2) \in \mathcal{L}_{\mathcal{H}}(EC)$, and w be a knowledge state. To prove that $p(e_1, e_2)$ holds in w, we must go through conditions *i*–*iv* of Definition 2.2. Step *i* costs $\mathcal{O}(1)$ $[\mathcal{O}(n^2)]$. Verifying the validity of propositions $e_1 \in [p|C\rangle$ and $e_2 \in \langle p|C]$ (conditions *ii* and *iii*, respectively) has constant cost $\mathcal{O}(1)$. Step *iv* consists of $\mathcal{O}(n)$ tests, each one of the same complexity of the test performed at step *i*, $\mathcal{O}(n)$ tests equal to that performed at step *ii*, and $\mathcal{O}(n)$ tests equal to that performed at step *iii*, and thus it costs $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$. This allows us to conclude that the complexity of model checking in *EC* is $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$.

CEC: $\mathcal{O}(kn)$ $[\mathcal{O}(kn^3)]$

A *CEC*-formula can be viewed as a boolean combination of *EC*-formulas. Therefore, checking a *CEC*-formula that contains k atomic formulas actually reduces to testing k *EC*-formulas. Since each test costs $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$, and the outcomes of the k tests can be combined in $\mathcal{O}(k)$ time to determine the truth value of the given formula, model checking in *CEC* costs $\mathcal{O}(k \cdot n)$ $[\mathcal{O}(k \cdot n^3)]$.

QEC: $\mathcal{O}(n^3)$ $[\mathcal{O}(n^5)]$

As we observed in Section 3.2, we can limit ourselves to *QEC*-formulas with at most two nested quantifiers. Let $\varphi = Q_1 x.\, Q_2 y.\, p(x, y)$ be such a formula with $Q_1, Q_2 \in \{\exists, \forall\}$. In the worst case, testing the truth of φ may require validating the *EC*-formulas $p(e_1, e_2)$, for every $e_1, e_2 \in \mathbf{E}$. Since checking an *EC*-formulas costs $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$, the complexity of model checking in *QEC* is $\mathcal{O}(n^2) \cdot \mathcal{O}(n) = \mathcal{O}(n^3)$ $[\mathcal{O}(n^2) \cdot \mathcal{O}(n^3) = \mathcal{O}(n^5)]$.

MEC: $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$

By Lemma 3.6, testing the truth of $\Diamond$- and $\Box$-moded atomic formulas reduces to testing the truth of local conditions. These conditions differ from those given in Definition 2.2 (intended model of *EC*) only for the replacement of some tests of the form $e_1 <_w e_2$ with other tests of the form $e_2 \not<_w e_1$. These changes do not affect the computational complexity of the testing procedure, and thus model checking in *EC* and in *MEC* have the same cost, i.e., $\mathcal{O}(n)$ $[\mathcal{O}(n^3)]$.

PEC: $\mathcal{O}(n^{3 \cdot B_{\mathcal{H}}+1})$ $[\mathcal{O}(n^{3 \cdot (B_{\mathcal{H}}+1)})]$

Let $\mathcal{H}$ be a *PEC*-structure, $p(e_1, e_2) \in \mathcal{L}_{\mathcal{H}}(PEC)$, and w be a knowledge state. To verify whether or not $p(e_1, e_2)$ holds in w we must go through steps *i*–*iv* of Definition 3.8.

We prove that model checking in *PEC* costs $Comp(B_{\mathcal{H}} = \mathcal{O}(n^{2 \cdot B_{\mathcal{H}}+1})$ $[= \mathcal{O}(n^{3 \cdot (B_{\mathcal{H}}+1)})]$ by induction on the value of $B_{\mathcal{H}}$:

- If $B_{\mathcal{H}} = 0$, model checking in *PEC* reduces to that in *EC*, and thus it costs $Comp(0) = \mathcal{O}(n)$ $[= \mathcal{O}(n^3)]$.
- When $B_{\mathcal{H}} > 0$, step *i* still costs $\mathcal{O}(1)$ $[\mathcal{O}(n^2)]$, while steps *ii* and *iii* become context dependent. Both of them involve the evaluation of at worst $D_{\mathcal{H}} \cdot C_{\mathcal{H}}$ preconditions. The evaluation of each precondition results in $\mathcal{O}(n^2)$ truth tests with nesting level $B_{\mathcal{H}} - 1$. Step *iv* involves $\mathcal{O}(n)$ tests equal to that of step *i*, $\mathcal{O}(n)$ tests equal to that of step *ii* and $\mathcal{O}(n)$ tests equal to that of step *iii*, and thus it results in $\mathcal{O}(n^3)$ $[\mathcal{O}(n^3)]$ tests with nesting level $B_{\mathcal{H}} - 1$. Hence, the complexity $Comp(B_{\mathcal{H}})$ can be expressed in terms of the complexity $Comp(B_{\mathcal{H}} - 1)$ by means of the recurrence expression $Comp(B_{\mathcal{H}}) = \mathcal{O}(n^3) \cdot Comp(B_{\mathcal{H}} - 1)$ $[Comp(B_{\mathcal{H}}) = \mathcal{O}(n^3) \cdot Comp(B_{\mathcal{H}} - 1)]$. By induction, it follows that $Comp(B_{\mathcal{H}}) = \mathcal{O}(n^{3 \cdot B_{\mathcal{H}}+1})$ $[Comp(B_{\mathcal{H}}) = \mathcal{O}(n^{3 \cdot (B_{\mathcal{H}}+1)})]$.

QMEC: $\mathcal{O}(n^3)$ $[\mathcal{O}(n^5)]$

The proof is similar to that for *QEC*.

CPEC: $\mathcal{O}(kn^{3 \cdot B_{\mathcal{H}}+1})$ $[\mathcal{O}(kn^{3 \cdot (B_{\mathcal{H}}+1)})]$

The proof is similar to that for *CEC*.

QPEC: $\mathcal{O}(n^{3 \cdot (B_{\mathcal{H}}+1)})$ $[\mathcal{O}(n^{3 \cdot (B_{\mathcal{H}}+1)+2})]$

The proof is similar to that for *QEC*. ■

In order to determine the complexity of model checking in *MPEC* and *QMPEC*, we first analyze the complexity of testing the truth of $\Diamond$-moded (resp. $\Box$-moded) atomic formulas. Let us call this problem the $\Diamond$-*MPEC* (resp. $\Box$-*MPEC*) problem.

Lemma 5.2 (*The complexity of the* $\Diamond$-*MPEC problem*)

The $\Diamond$-MPEC *problem is* **NP**-*complete.*

Proof.

We first prove that $\Diamond$-*MPEC* is in **NP**. Let ψ be an atomic formula and w a knowledge state. Any "yes" instance of the problem has a *succinct certificate* of its being a "yes" instance and this certification has polynomial time complexity. The certificate is the extension w' of w in which ψ holds. Any "yes" instance has this certificate. The test $w' \models \psi$ is polynomial by Theorem 5.1.

In order to prove that the considered problem is **NP**-hard, we define a (polynomial) reduction of *3SAT* [Pap94] into the $\Diamond$-*MPEC* problem.

Let q be a boolean formula in *3CNF*, $p_1, p_2, \ldots$, and p_n be the propositional variables that occur in q, and $q = c_1 \wedge c_2 \wedge \ldots \wedge c_m$, where $c_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$ and, for each $l_{i,j}$, either $l_{i,j} = p_k$ or $l_{i,j} = \neg p_k$, for some k.

We define a *PEC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot],)$ such that:

$$\mathbf{E} = \{e_1^i, e_2^i : 1 \leq i \leq n\} \cup \{e_1(c_i), e_2(c_i) : 1 \leq i \leq m\} \cup \{e_1(q), e_2(q)\};$$
$$\mathbf{P} = \{p_i, \overline{p}_i : 1 \leq i \leq n\} \cup \{c_i : 1 \leq i \leq m\} \cup \{q\},$$

and the context-dependent initiating and terminating maps are defined as follows:

- for any $1 \leq i \leq n$, $[p_i|\{\}\rangle = \langle \overline{p}_i|\{\}] = \{e_1^i\}$ and $[\overline{p}_i|\{\}\rangle = \langle p_i|\{\}] = \{e_2^i\}$;
- for any $1 \leq i \leq m$, $1 \leq j \leq 3$, $[c_i|\{l'_{i,j}\}\rangle = \{e_1(c_i)\}$ and $\langle c_i|\{\}] = \{e_2(c_i)\}$, where for each i, j, if, for some k, $l_{i,j} = p_k$, then $l'_{i,j} = p_k$; otherwise $(l_{i,j} = \neg p_k)$ $l'_{i,j} = \overline{p}_k$;
- $[q|\{c_1, c_2 \ldots c_m\}\rangle = \{e_1(q)\}$ and $\langle q|\{\}] = \{e_2(q)\}$.

Moreover, let $w = \emptyset$ and $\psi = c'_1 \wedge c'_2 \wedge \ldots \wedge c'_m$, where $c'_i = l'_{i,1} \vee l'_{i,2} \vee l'_{i,3}$, and, for each i, j, if $l_{i,j} = p_k$, then $l'_{i,j} = p_k(e_1^k, e_2^k)$; otherwise $(l_{i,j} = \neg p_k)$ $l'_{i,j} = \overline{p}_k(e_2^k, e_1^k)$.

We have that $w \models \Diamond\psi$ if and only if q is satisfiable. ■

Since $\Box = \neg\Diamond\neg$, it easily follows from Lemma 5.2 that the problem of testing whether an atomic formula is necessarily true in *MPEC* is **coNP**-complete.

Corollary 5.3 (*The complexity of the $\Box$-MPEC problem*)

The $\Box$-MPEC problem is **coNP***-complete.* ■

Theorem 5.4 (*Event calculi in* $\mathbf{P^{NP}}$)

Model checking in MPEC *and* QMPEC *is in* $\mathbf{P^{NP}}$*, and it is both* **NP**- *and* **coNP***-hard.*

Proof.

We first prove that model checking in *MPEC* and *QMPEC* is in $\mathbf{P^{NP}}$.

Let φ be a *MPEC*-formula and w a knowledge state. If φ is a propositional formula, then model checking has polynomial cost by virtue of Theorem 5.1; if φ is a $\Diamond$-moded atomic formula, then verifying $w \models \varphi$ is **NP**-complete by Lemma 5.2; finally, if φ is a $\Box$-moded atomic formula, then Lemma 5.3 proves that testing $w \models \varphi$ is **coNP**-complete. Thus, the whole problem of testing $w \models \varphi$ for *MPEC* involves either a polynomial check, or an **NP**-check, or a **coNP**-check. This means that it can be computed by a Turing machine which can access an **NP**-oracle and runs in deterministic polynomial time, and hence the problem is in $\mathbf{P^{NP}}$. Since only one call to the oracle is needed, it is actually in $\mathbf{P^{NP[1]}}$.

Any *QMPEC*-formula may have at most two nested quantifiers. Hence, model checking for a formula of *QMPEC* involves at most $\mathcal{O}(n^2)$ evaluations of *MPEC*-formulas. As we have just shown, each of these evaluations is in $\mathbf{P^{NP[1]}}$, and thus model checking in *QMPEC* is in $\mathbf{P}^{\mathbf{NP}[n^2]}$. ■

The hardness results follow from Lemma 5.2 and Corollary 5.3.

Theorem 5.5 (**PSPACE**-*complete event calculi*)

Model checking in QCEC, CMEC, QCMEC, QCPEC, CMPEC, *and* QCMPEC *is* **PSPACE***-complete.*

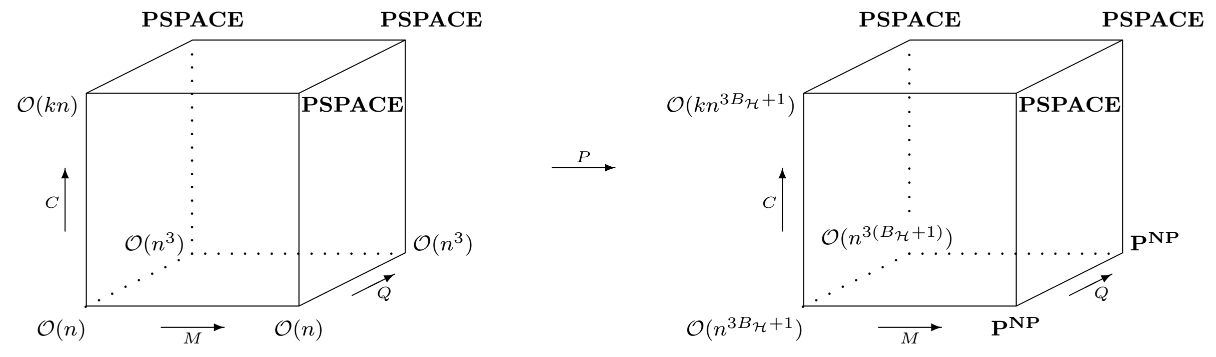

Figure 5: The Complexity of the *EC* cube

Proof.

It suffices to prove that (i) model checking in *QCEC* and *CMEC* is **PSPACE**-hard and (ii) model checking in *QCMPEC* is polynomial-space bounded. The proof of **PSPACE**-hardness for *QCEC* and *CMEC* can be found in [CFM98b] and [CM99], respectively.

In order to prove that the problem of model checking in *QCMPEC* is in **PSPACE**, we show that this problem belongs to **AP**, that is, we define an alternating polynomial time algorithm that solves it. Let φ be a *QCMPEC*-formula and w a knowledge state. If $\varphi = \alpha \wedge \beta$ (resp. $\varphi = \alpha \vee \beta$), the algorithm enters in an AND (resp. OR) state. It nondeterministically chooses one among α and β and evaluates it in w. If $\varphi = \neg(\alpha \wedge \beta)$ (resp. $\varphi = \neg(\alpha \vee \beta)$), the algorithm evaluates $\neg\alpha \vee \neg\beta$ (resp. $\neg\alpha \wedge \neg\beta$). If $\varphi = \neg\neg\alpha$, the algorithm verifies α. If $\varphi = \Box\alpha$ (resp. $\varphi = \Diamond\alpha$), the algorithm enters in an AND (resp. OR) state. It nondeterministically chooses one extension w' of w and evaluates α in w'. If $\varphi = \neg\Box\alpha$ (resp. $\varphi = \neg\Diamond\alpha$), the algorithm evaluates $\Diamond\neg\alpha$ (resp. $\Box\neg\alpha$). If $\varphi = \forall x.\, \alpha$ (resp. $\varphi = \exists x.\, \alpha$), the algorithm enters in an AND (resp. OR) state. It nondeterministically chooses one event, say e, and evaluates the formula obtained by replacing all occurrences of x in α that are in the scope of the quantifier by e. If $\varphi = \neg\forall x.\, \alpha$ (resp. $\varphi = \neg\exists x.\, \alpha$), the algorithm evaluates $\exists x.\, \neg\alpha$ (resp. $\forall x.\, \neg\alpha$). If $\varphi = p(e_1, e_2)$ (resp. $\varphi = \neg p(e_1, e_2)$), the algorithm accepts it if and only if all conditions (resp. at least one condition) from *i* to *iv* of Definition 3.8 hold (resp. do not hold).

From the definition of acceptance of alternating machines [Pap94], it follows that an *QCMPEC*-instance $(\mathcal{H}, \varphi, w)$ is accepted if and only if $I_{\mathcal{H}}; w \models \varphi$. Moreover, the time needed is polynomial in the size of $\mathcal{H}$ and φ. Thus, model checking in *EQCMEC* is in **AP**. Since **AP** = **PSPACE** [Pap94], it is in **PSPACE**. ■

It is worth noting that the (deterministic) time complexity of the model checking procedure we exploited in the proof of Theorem 5.5 is exponential in the query complexity (length of the formula) for event calculi provided with quantifiers, but devoid of modalities (*QCEC* and *QCPEC*), it is exponential in the data complexity (number of events) for event calculi with modalities, but devoid of quantifiers (*CMEC* and *CMPEC*), and it is exponential in both the data and query complexities for event calculi with both modalities and quantifiers (*QCMEC* and *QCMPEC*). In most problems of interest, we need to deal with situations where there is a large number of events, but the size of relevant queries is generally limited. The examples given in Section 4 fall into this category. In such a case, the fact that a calculus is polynomial in the number of events is essential, because being the exponent dependent on the length of the formula may at worst lead to polynomials of high degree.

These complexity results are summarized in Figure 5, which is isomorphic to Figure 1.

6 Approximate Event Calculi

The complexity results given in Section 5 allow us to conclude that model checking in the calculi analyzed in Theorems 5.4 and 5.5 (probably) involves an exponential-time cost in the data complexity (number of events) and/or in the query complexity (size of the formula). As already pointed out in Section 5, data complexity is the critical factor in problems of practical relevance. In this section, we present approximate model checking procedures, that are in many cases either sound (but not complete) or complete (but not sound) with respect to the semantics of the corresponding event calculi, but which are polynomial at least in the number of events.

We first consider the case of $\Box$- or $\Diamond$-moded atomic formulas. In the context-independent case, Lemma 3.6 guarantees the existence of a polynomial time algorithm to test the truth of such formulas. When considering preconditions, the problem of checking a $\Box$-moded (resp. $\Diamond$-moded) atomic formula is complete in **NP** (resp. **coNP**) as shown in Lemma 5.2 (resp. Corollary 5.3). Hence, it is unlikely that there exists a polynomial algorithm that computes exactly the set of necessary and possible *MVI*s.

In the following, we present a polynomial algorithm that *approximates* the computation of necessary and possible *MVI*s in the context-dependent case. In order to make the notion of approximation more precise, we rely on the sets $MVI(w)$, $\Box MVI(w)$ and $\Diamond MVI(w)$, defined as follows:

$$\begin{array}{rcl} MVI(w) & = & \{p(e_1,e_2) : w \models p(e_1,e_2)\}; \\ \Box MVI(w) & = & \{p(e_1,e_2) : w \models \Box p(e_1,e_2)\}; \\ \Diamond MVI(w) & = & \{p(e_1,e_2) : w \models \Diamond p(e_1,e_2)\}. \end{array}$$

They respectively denote the sets of MVIs, necessary MVIs, and possible MVIs with respect to w. They can be viewed as functions of the knowledge state w.

The basic idea will be to compute, in polynomial time, useful *bounds* on $\Box$MVI$(\cdot)$ and $\Diamond$MVI$(\cdot)$, that is, subsets of $\Box$MVI$(\cdot)$ and supersets of $\Diamond$MVI$(\cdot)$. Both functions have a trivial bound, as shown by the following inclusion chain [CM99]:

$$\emptyset \quad \subseteq \quad \Box\text{MVI}(\cdot) \quad \subseteq \quad \text{MVI}(\cdot) \quad \subseteq \quad \Diamond\text{MVI}(\cdot) \quad \subseteq \quad \mathcal{L}(PEC) = \mathcal{L}(EC).$$

To identify useful bounds, we generalize the local conditions of Lemma 3.6 to deal with preconditions. Such a generalization yields the definition of two meta-predicates, *necHolds* and *posHolds*, which specify local conditions for $\Box$- and $\Diamond$-moded atomic formulas with non-empty contexts, respectively.

Definition 6.1 (*The* necHolds *and* posHolds *meta-predicates*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$ be a PEC-structure. Given two events $e_1, e_2 \in \mathbf{E}$, a property $p \in \mathbf{P}$, and a state of knowledge $w \in W_{\mathcal{H}}$, the meta-predicates necHolds and posHolds are mutually defined as follows:

$necHolds(p, e_1, e_2, w)$ *iff*

i. $e_1 <_w e_2$

ii. $necInit(e_1, p, w)$, *where* $necInit(e_1, p, w)$ *iff*

$\exists C \in \mathbf{2^P}.\ e_1 \in [p|C\rangle \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.\ necHolds(q, e', e'', w) \quad \wedge \quad e' <_w e_1 \quad \wedge \quad e_1 \leq_w e''$

iii. $necTerm(e_2, p, w)$, *where* $necTerm(e_2, p, w)$ *iff*

$\exists C \in \mathbf{2^P}.\ e_2 \in \langle p|C] \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.\ necHolds(q, e', e'', w) \quad \wedge \quad e' <_w e_2 \quad \wedge \quad e_2 \leq_w e''$

iv. $\neg necBroken(p, e_1, e_2, w)$, *where* $necBroken(p, e_1, e_2, w)$ *iff*

$\exists e \in \mathbf{E}.\ e \not\leq_w e_1 \quad \wedge \quad e_2 \not\leq_w e \quad \wedge \quad (posInit(e, p, w) \ \vee\ posTerm(e, p, w))$

posHolds(p, e_1, e_2, w) *iff*

i. $e_2 \not<_w e_1$

ii. *posInit*(e_1, p, w), *where posInit*(e_1, p, w) *iff*
$\exists C \in \mathbf{2^P}.\ e_1 \in [p|C\rangle \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.\ posHolds(q, e', e'', w) \quad \wedge \quad e_1 \not\leq_w e' \quad \wedge \quad e'' \not<_w e_1$

iii. *posTerm*(e_2, p, w), *where posTerm*(e_2, p, w) *iff*
$\exists C \in 2^{\mathsf{P}}.\ e_2 \in \langle p|C] \quad \wedge \quad \forall q \in C.\ \exists e', e'' \in \mathbf{E}.\ posHolds(q, e', e'', w) \quad \wedge \quad e_2 \not\leq_w e' \quad \wedge \quad e'' \not<_w e_2$

iv. $\neg$*posBroken*(p, e_1, e_2, w), *where posBroken*(p, e_1, e_2, w) *iff*
$\exists e \in \mathbf{E}.\ e_1 <_w e \quad \wedge \quad e <_w e_2 \quad \wedge \quad (necInit(e, p, w) \ \vee \ necTerm(e, p, w))$ $\square$

These definitions merge the contents of Lemma 3.6 and Definition 3.8. In particular, they differ from the latter only by the replacement of conditions of the form $e <_w e'$ with their negation or with the symmetric condition.

The following theorem proves that the meta-predicates *necHolds* and *posHolds* respectively compute a subset of $\square$MVI$(\cdot)$ and a superset of $\Diamond$MVI$(\cdot)$. Moreover, it is possible to show that if there are no preconditions (context-independent case), they compute exactly $\square$MVI$(\cdot)$ and $\Diamond$MVI$(\cdot)$, respectively.

Theorem 6.2 (*Approximation procedures for the $\square$-MPEC and $\Diamond$-MPEC problems*)

Let $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$ be a MPEC-structure. Given $e_1,\ e_2 \in \mathbf{E}$, $p \in \mathbf{P}$, and $w \in W_{\mathcal{H}}$, it holds that

i. *if* $necHolds(p, e_1, e_2, w)$, *then* $w \models \square p(e_1, e_2)$;

ii. *if* $w \models \Diamond p(e_1, e_2)$, *then* $posHolds(p, e_1, e_2, w)$.

Proof.

We only outline the structure of the proof. A rigorous proof is simple, but long and somewhat tedious.

Since the meta-predicates *necHolds* and *posHolds* are defined in term of each other, we prove the statements *i* and *ii* by mutual induction on the length $B_{\mathcal{H}}$ of the longest path on the dependency graph for $\mathcal{H}$. If $B_{\mathcal{H}} = 0$, the thesis easily follows from Definition 6.1 and Lemma 3.6.

Assume then that $B_{\mathcal{H}} > 0$. The following statements hold:

(1)	if	$e_1 <_w e_2$,	then	$\forall w' \in Ext(w).\ e_1 <_{w'} e_2$
(2)	if	$necInit(e_1, p, w)$,	then	$\forall w' \in Ext(w).\ init(e_1, p, w')$
(3)	if	$necTerm(e_2, p, w)$,	then	$\forall w' \in Ext(w).\ term(e_2, p, w')$
(4)	if	$\neg necBroken(p, e_1, e_2, w)$,	then	$\forall w' \in Ext(w).\ \neg br(p, e_1, e_2, w')$

The proof of (1) is simple. Statements (2) and (3), and statement (4) can be proved by exploiting the induction hypothesis on *i* and *ii*, respectively. The conjunction of the antecedents of (1), (2), (3), and (4) defines the meta-predicate *necHolds*(p, e_1, e_2, w), while the conjunction of the consequents of (1), (2), (3), and (4) defines the semantics of $w \models \square p(e_1, e_2)$. Therefore, if *necHolds*(p, e_1, e_2, w), then $w \models \square p(e_1, e_2)$.

Analogously, we have the following statements:

(5)	if	$\exists w' \in Ext(w).\ e_1 <_{w'} e_2$,	then	$e_2 \not<_w e_1$
(6)	if	$\exists w' \in Ext(w).\ init(e_1, p, w')$,	then	$posInit(e_1, p, w)$
(7)	if	$\exists w' \in Ext(w).\ term(e_2, p, w')$,	then	$posTerm(e_2, p, w)$
(8)	if	$\exists w' \in Ext(w).\ \neg br(p, e_1, e_2, w')$,	then	$\neg posBroken(p, e_1, e_2, w)$

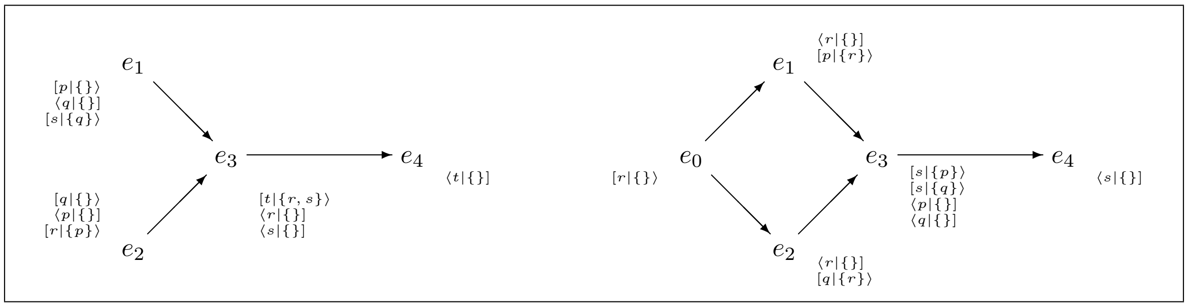

Figure 6: Approximations of Modal Event Calculi with Preconditions

The proof of (5) is simple. We can prove statements (6) and (7), and statement (8) by exploiting the induction hypothesis on *i* and *ii*, respectively. If $w \models \Diamond p(e_1, e_2)$, then the conjunction of the antecedents of (5), (6), (7), and (8) holds. Moreover, the conjunction of the consequents of (5), (6), (7), and (8) defines the meta-predicate $posHolds(p, e_1, e_2)$. Therefore, if $w \models \Diamond p(e_1, e_2)$, then $posHolds(p, e_1, e_2, w)$. ■

We will now show that the inclusions of Theorem 6.2 are strict. The first example (Figure 6, left side) models the following situation. Let e_1, e_2, e_3, and e_4 be four events, and p, q, r, s, and t be five properties. Suppose that e_1 initiates p and terminates q, without preconditions, and it initiates s, with precondition q; e_2 initiates q and terminates p, without preconditions, and it initiates r, with precondition p; e_3 terminates both r and s, without preconditions, and it initiates t, with preconditions r and s; finally, e_4 terminates t, without preconditions. Consider a knowledge state w according to which e_1 precedes e_3, e_2 precedes e_3, the relative order of e_1 and e_2 is unknown, and e_3 precedes e_4. Under such hypotheses, $posHolds(t, e_3, e_4, w)$ is true, but $w \not\models \Diamond t(e_3, e_4)$. The second example (Figure 6, right side) describes a dual situation. It is not difficult to verify that $w \models \Box s(e_3, e_4)$, but $necHolds(s, e_3, e_4, o^+)$ is false.

The complexity of the procedures *necHolds* and *posHolds* can be easily shown to be polynomial in the number of events. The semantics of the meta-predicates *necHolds* and *posHolds* indeed differ from the intended *PEC*-model (Definition 3.8) only for the fact that some tests of the form $e_1 <_w e_2$ are replaced by tests of the form $e_2 \not<_w e_1$. This replacement does not affect the overall complexity; thus the cost of computing the meta-predicates *necHolds* and *posHolds* is equal to the cost of model checking in *PEC* (cf. Theorem 5.1). We use the same conventions as in Section 5.

Corollary 6.3 (*Complexity of MPEC approximations*)

The cost of the meta-predicates necHolds and posHolds is $\mathcal{O}(n^{2\cdot B_{\mathcal{H}}+1})$ $[\mathcal{O}(n^{3\cdot(B_{\mathcal{H}}+1)})]$. ■

It is straightforward to extend the above approximation procedures to deal with *QMPEC*-formulas. The complexity of the resulting procedures is equal to the complexity of model checking in *QPEC*, i.e., $\mathcal{O}(n^{2\cdot B_{\mathcal{H}}+3})$ $[\mathcal{O}(n^{3\cdot(B_{\mathcal{H}}+1)+2})]$.

We now turn to our most general event calculus, *QCMPEC*. We want to identify general classes of formulas that enjoy approximations that are either sound (but not necessarily complete) or complete (but not necessarily sound), in the same sense as for atomic modal formulas above.

To this effect, we can try to take advantage of some logical equivalences that hold in *QCMPEC* (e.g. those in Proposition 3.10) to push modalities as close to the atomic subformulas as possible, in order to apply the approximation procedure *necHolds* and *posHolds* to $\Box$-moded and $\Diamond$-moded atomic formulas, respectively. Unfortunately, this goal cannot be always successfully accomplished.

In particular, there is no general way of reducing formulas of the forms $\Box(\varphi_1 \vee \varphi_2)$, $\Box\exists X\varphi(X)$, $\Diamond(\varphi_1 \wedge \varphi_2)$, and $\Diamond\forall X\varphi(X)$. Moreover, formulas of the forms $\Box\Diamond\varphi$ and $\Diamond\Box\varphi$ can be reduced only for some classes of argument formulas φ. To deal with these critical cases, we exploit the following logical implications between *QCMPEC*-formulas

Proposition 6.4 (*QCMPEC logical implications*)

Let φ, φ_1, and φ_2 be QCMPEC-formulas, and $p(e_1, e_2)$ be an atomic formula. For every knowledge state $w \in W$, it holds that

$$\begin{array}{rlcl}
i. & w \models (\Box\varphi_1 \vee \Box\varphi_2) & \Rightarrow & w \models \Box(\varphi_1 \vee \varphi_2) \\
ii. & w \models \exists X\Box\varphi(X) & \Rightarrow & w \models \Box\exists X\varphi(X) \\
iii. & w \models \Box p(e_1, e_2) & \Rightarrow & w \models \Box\Diamond p(e_1, e_2)
\end{array}$$

and

$$\begin{array}{rlcl}
iv. & w \models \Diamond(\varphi_1 \wedge \varphi_2) & \Rightarrow & w \models (\Diamond\varphi_1 \wedge \Diamond\varphi_2) \\
v. & w \models \Diamond\forall X\varphi(X) & \Rightarrow & w \models \forall X\Diamond\varphi(X) \\
vi. & w \models \Diamond\Box p(e_1, e_2) & \Rightarrow & w \models \Diamond p(e_1, e_2)
\end{array}$$

■

Now consider a *QCMPEC*-formula $\varphi^\Box$ that does contain neither $\Diamond$, nor $\neg$ (once implications have been expanded). Then, we can apply the logical equivalences in Proposition 3.10 and the logical implications *i*–*ii* in Proposition 6.4 (backwards) to push modal operators inside $\varphi^\Box$. Observe that this procedure terminates with a formula $\phi^\Box$ where all $\Box$ are applied to atomic formulas only, that it does not introduce any $\Diamond$ (only pushing a $\Box$ inside a negation can produce a $\Diamond$), and that the resulting chain of formulas is sound with respect to $\varphi^\Box$, but not necessarily complete. This means that if $\phi^\Box$ is provable (using *necHolds* as a test for $\Box$-moded atoms) so is $\varphi^\Box$, but that $\varphi^\Box$ could be true even if $\phi^\Box$ does not hold.

A dual situation holds for negation-free *QCMPEC*-formulas $\varphi^\Diamond$ that may contain $\Diamond$, but that do not mention $\Box$. Now, applying Proposition 3.10 and the logical implications *iv*–*v* of Proposition 6.4 (forward), we obtain a formula $\phi^\Diamond$, where $\Diamond$ appears in front of atoms only, that is complete but not necessarily sound with respect to $\varphi^\Diamond$: if $\phi^\Diamond$ is not provable (using *posHolds* as a test for $\Diamond$-moded atoms) neither is $\varphi^\Diamond$, but that $\varphi^\Diamond$ could be false even if $\phi^\Diamond$ holds.

We call the procedure we just outlined *approx.* The classes of formulas it handles can be slightly extended to admit negation outside of modalities, or formulas where one of the modalities always clings to an atomic formula and is preceded by the other (we can then use the implications *iii* and *vi* of Proposition 6.4 to get rid of it). In general, any formula whose reduction and validity check does not mix unsound and incomplete steps is acceptable. Notice that the formulas appearing in all of our case studies in Section 4 satisfy this criterion. We continue to use $\varphi^\Box$ and $\varphi^\Diamond$ for formulas of the two extended classes. When applied to an arbitrary *QCMPEC*-formula, *approx* does not maintain provability: is is neither sound nor complete in general. Nevertheless, it can actually serve as a useful approximation in many concrete cases. We implement this more general procedure in Section 7.

It is possible to show that applying the procedure *approx* and evaluating the resulting formula is polynomial-time bound in the data complexity: (i) the replacement of the original formula φ^* (for $* \in \{\Box, \Diamond\}$) by the formula ϕ^* takes polynomial time in the length of φ^*, and constant time in the number of events; (ii) model checking ϕ^* in *QCPEC* is polynomial in the data complexity; (iii) the approximation procedures *necHolds* and *posHolds* have polynomial cost in the number of events by virtue of Corollary 6.3. Furthermore, by Theorem 6.2, *approx* is sound for formulas $\varphi^\Box$ (resp. complete for formulas $\varphi^\Diamond$), that is, if *approx* yields reduced formulas $\phi^\Box$ (resp. $\phi^\Diamond$), we have that if $w \models \phi^\Box$ then $w \models \varphi^\Box$ (resp. if $w \models \varphi^\Diamond$ then $w \models \phi^\Diamond$).

The procedure *approx* applies transparently to approximately checking basic modalities in the subcalculi *CMEC*, *CMPEC*, and *QCMEC*. In particular, the resulting validity test for *CMEC* and *CMPEC* are polynomial in both data and query complexities.

7 Implementation

The Event Calculus [KS86] has traditionally been implemented in *Prolog*. In recent years, we have instead investigated the use the language of hereditary Harrop formulas [MNPS91] and its concrete realization as the logic programming language λ*Prolog* [Mil96], which declaratively extends *Prolog* with a number of constructs. This has enabled us to achieve declarative yet simple encodings of various extensions of the event calculus [CFM97b, CFM98c, CM99]. Similar attempts using *Prolog* produces complex encodings [CMP94] that do not lend themselves to formally establishing correctness issues. In [CM99], we instead proved the soundness and completeness of an encoding of *CMEC* (then called *GMEC*) as a program in the language of hereditary Harrop formulas, with respect to the semantic rules outlined in the previous sections.

In this section, we use λ*Prolog* and its module facilities to achieve an economical implementation of all the calculi discussed in this paper. We also summarize relevant correctness statements.

7.1 λ*Prolog* in a Nutshell

We shall assume the reader familiar with the logic programming language *Prolog*, for which we adopt [SS94] as a reference. We will instead illustrate some of the characteristic constructs of λ*Prolog* at an intuitive level. We invite the interested reader to consult [Mil96] for a more complete discussion of this language, and [CM99] for a detailed presentation in the context of the Event Calculus.

Differently from *Prolog* which is first-order, λ*Prolog* is a *higher-order* language, which means that the terms in this programming language are drawn from a *simply typed* λ*-calculus*. More precisely, the syntax of terms is given by the following grammar:

$$M \ ::= \ c \ | \ x \ | \ F \ | \ M_1\, M_2 \ | \ x \setminus M$$

where c ranges over *constants*, x stands for a *bound variable* and F denotes a *logical variable* (akin to *Prolog*'s variables). Identifiers beginning with a lowercase and an uppercase letter stand for constants and logical variables, respectively. Terms that differ only by the name of their bound variables are considered indistinguishable. "$x \setminus M$" is λ*Prolog*'s syntax for λ*-abstraction*, traditionally written $\lambda x.\, M$. In this language, terms and atomic formulas are written in curried form (e.g. "`before E1 E2`" rather than "`before(E1, E2)`"). Syntax is provided for declaring infix symbols. We will rely on two predefined symbols: the infix list constructor, written "`::`", and the empty list, denoted "`nil`".

Every constant, bound variable and logical variable is given a unique *type* A. Types are either user-defined *base types*, or *functional types* of the form A_1 `->` A_2. By convention, the predefined base type `o` classifies formulas. A base type a is declared as "`kind` a`.`", and a constant c of type A is entered in λ*Prolog* as "`type` c A`.`". Application and λ-abstraction can be typed if their subexpression satisfy certain constraints. A list of elements of type A has predeclared type "`list` A". λ*Prolog* will reject every term that is not typable.

While first-order terms are equal solely to themselves, the equational theory of higher-order languages identifies terms that can be rewritten to each other by means of the β*-reduction* rule: $(x \setminus M)\, N = [N/x]M$, where the latter expression denotes the capture-avoiding substitution of the term N for the bound variable x in M. A consequence of this fact is that unification in λ*Prolog* must perform β-reduction on the fly in order to equate terms or instantiate logical variables. A further difference from *Prolog* is that logical variables in λ*Prolog* can stand for functions (i.e. expressions of the form $x \setminus M$) and this must be taken into account when unification is performed.

For our purposes, the language of formulas of *λProlog* differs from *Prolog* for the availability of implication, intensional universal quantification, and of an explicit existential quantifier in the body of clauses. The goal $D \supset G$, written "D `=>` G" in the concrete syntax of this language, is solved by resolving the goal G after having assumed D as an additional program clause. Solving the goal $\forall x.\, G$, denoted "`pi` $x \setminus G$" in the concrete syntax, amounts to inventing a new constant c of the appropriate type and finding a solution to $[c/x]G$. Finally, the goal $\exists x.\, G$ is entered as "`sigma` $x \setminus G$". We will also take advantage of *negation-as-failure*, denoted `not`. Other connectives are denoted as in *Prolog*: "`,`" for conjunction, "`;`" for disjunction, "`:-`" for implication with the arguments reversed. The only predefined predicate we will use is the infix "`=`" that unifies its arguments. Given a well-typed *λProlog* program $\mathcal{P}$ and a goal G, the fact that there is a derivation of G from $\mathcal{P}$, i.e. that G is solvable in $\mathcal{P}$, is denoted $\mathcal{P} \vdash G$. See [CM99, Mil96] for details.

λProlog offers also the possibility of organizing programs into modules. A module m is declared as "`module` m`.`" followed by the declarations and clauses that define it. Modules can access other modules by means of the `accumulate` declaration. Whenever "`accumulate` m_1`.`" occurs in the preamble of a module m_2, it specifies a module consisting of all the clauses of m_1 followed by all the clauses of m_2.

Finally, `%` starts a comment that extends to the end of the line.

7.2 Encoding

We will now give a *λProlog* implementation of *EC* and of the extensions discussed above. We report the resulting code in Appendix A as a collection of modules named after the corresponding calculi. An encoding of the case studies presented in Section 4 can be found in Appendix B. This code has been tested using the *Terzo* implementation of *λProlog*, version 1.2b, which is available from `http://www.cse.psu.edu/~dale/lProlog/`.

We now define a family of representation functions $\ulcorner\cdot\urcorner$ that relate the mathematical entities we have been using in Sections 2, 3, and 6 to terms in *λProlog*. Specifically, we need to encode *EC*-structures, the associated orderings, and the language of each of our calculi.

Orderings

For implementation purposes, it is more convenient to compute the relative ordering of two events on the basis of fragmented data (a binary acyclic relation) than to maintain this information as a strict order. We rely on the binary predicate symbol `beforeFact` to represent the edges of the binary acyclic relation. We encapsulate the clauses for the predicate `before`, which implements its transitive closure, in the module `ordering`, shown in Appendix A.1. We have proved in [CMC95] that, in pathological cases, this code can establish an ordering relation in a time exponential in the number of events. A quadratic implementation can however be found in [CMC95].

EC-structures and MVIs

We represent a generic *EC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ by giving an encoding of the entities that constitute it. We introduce the types `event` and `property` so that every event in $\mathbf{E}$ (property in $\mathbf{P}$) is represented by a distinct constant of type `event` (of type `property`). Event variables are represented as *λProlog* variables of the relative type. The initiation, termination and exclusivity relations, and event occurrences (traditionally represented in *EC*) are mapped to the predicate symbol `initiates`, `terminates`, `exclusive`, and `happens`, respectively, applied to the appropriate arguments. Declarations for these constants can be found in Appendix A.2.

In order to encode the syntax of *EC* and of its extensions, we define the type `mvi`, intended to represent the formulas of those language (as opposed to the formulas of *λProlog*, that have predefined type `o`). We then represent an atomic formula to be tested for MVI-hood by means of the function symbol `period`:

$$\ulcorner p(e_1, e_2)\urcorner = \texttt{period}\,\ulcorner e_1\urcorner\,\ulcorner p\urcorner\,\ulcorner e_2\urcorner$$

The truth of a formula is expressed by means of the predicate `holds`, which accepts an object of type `mvi` as an argument. These declarations have been collected in the module `mvis` in Appendix A.2.

The predicates `init`, `term`, and `excl` have the purpose of providing a uniform interface to initiation, termination and exclusivity relations, both in the presence and in the absence of preconditions. We will describe them in more detail when illustrating the encoding of *EC* and of preconditions.

Connectives

The syntax and semantics of the boolean connectives can be introduced independently from the above formalization of *EC* (except for having them operate on expressions of type `mvi`). We represent them by means of the constants `neg`, `and`, `or`, and `implies`:

$$\begin{aligned}
\ulcorner\neg\varphi\urcorner &= \texttt{neg}\,\ulcorner\varphi\urcorner\\
\ulcorner\varphi_1 \wedge \varphi_2\urcorner &= \ulcorner\varphi_1\urcorner\,\texttt{and}\,\ulcorner\varphi_2\urcorner\\
\ulcorner\varphi_1 \vee \varphi_2\urcorner &= \ulcorner\varphi_1\urcorner\,\texttt{or}\,\ulcorner\varphi_2\urcorner\\
\ulcorner\varphi_1 \rightarrow \varphi_2\urcorner &= \ulcorner\varphi_1\urcorner\,\texttt{implies}\,\ulcorner\varphi_2\urcorner
\end{aligned}$$

Clauses `CONN-1` to `CONN-4` in module `connectives` (Appendix A.3) reduce the truth check for the boolean connectives to the derivability of the corresponding *λProlog* constructs. Notice that implication is translated back to a combination of negation and disjunction in clause `CONN-4`. This module implements the vertical edges in Figure 1.

Quantifiers

Quantifiers differ from the other syntactic entities of a language by the fact that they *bind* a variable in their argument (e.g. x in $\exists x.\,\varphi$). Bound variables are then subject to implicit renaming to avoid conflicts and to substitution. Encoding binding constructs in traditional programming languages such as *Prolog* is painful since these operations must be explicitly programmed. *λProlog* and other higher-order languages permit a much leaner emulation since λ-abstraction ($x \setminus M$) is itself a binder and their implementations come equiped with (efficient) ways of handling it. The idea, known as *higher-order abstract syntax* [Mil96], is then to use *λProlog*'s abstraction mechanism as a universal binder. Binding constructs in the object language are then expressed as constants that takes a λ-abstracted term as its argument. The quantified formulas of our calculus are indeed represented as follows:

$$\begin{aligned}
\ulcorner\forall x.\,\varphi\urcorner &= \texttt{forAllEvent}\ (x \setminus \ulcorner\varphi\urcorner)\\
\ulcorner\exists x.\,\varphi\urcorner &= \texttt{forSomeEvent}\ (x \setminus \ulcorner\varphi\urcorner)
\end{aligned}$$

Both `forAllEvent` and `forSomeEvent` are declared of type `(event -> mvi) -> mvi` in Appendix A.4. Variable renaming happens behind the scenes, and substitution is delegated to the meta-language as β-reduction.

An example will shed some light on this technique. Consider the formula $\varphi = \exists x.\,p(x, e_2)$, whose representation is

```
forSomeEvent (x \ (period x p e2))
```

where we have assumed that p and e_2 are encoded as the constants `p` and `e2`, of the appropriate type. It is easy to convince oneself that this expression is well-typed. In order to ascertain the truth of φ, we need to check whether $p(e, e_2)$ holds for successive $e \in E$ until such an event is found. Automating this implies that, given a candidate event e_1 (represented as `e1`), we need to substitute `e1` for `x` in `period x p e2`. This can however be achieved by simply applying the argument of `forSomeEvent` to `e1`. Indeed, `(x \ (period x p e2)) e1` is equal to `period e1 p e2`, modulo β-reduction. This technique is used in Appendix A.4, that contains the code implementing quantifiers.

Although λ*Prolog* offers a form of universal quantification, we are forced to take a detour and express our universal quantifiers as negations and existentials in clause `QUANT-1`. A lengthy discussion of the logical reasons behind this step can be found in [CM99]. Existential quantification is instead mapped to the corresponding λ*Prolog* construct in clause `QUANT-2`. Module `quantifiers` implements the oblique edges in Figure 1.

Generic Modalities

A implementation of the unrestricted modal operators $\Box$ and $\Diamond$ is contained in the module `modalities`, displayed in Appendix A.5. With the exception of *MEC*, it implements the horizontal edges in Figure 1. A specialized (and more efficient) code in the case of *MEC* and the approximate calculi is discussed below. Modal formulas are represented below by means of the constants `must` and `may`:

$$\ulcorner \Box\varphi \urcorner = \texttt{must}\ \ulcorner \varphi \urcorner$$
$$\ulcorner \Diamond\varphi \urcorner = \texttt{may}\ \ulcorner \varphi \urcorner$$

The clauses `MOD-1` to `MOD-4` implement the Unfolding Lemma proved in [CM99]. Intuitively, this results states that the truth test for an arbitrary formula having a modality as its main connective can be reduced to first testing the truth of its immediate subformula in the current world and then checking the truth of the original formula in the 'one-step' extensions of the current knowledge state.

These clauses are interesting since they make use of an additional construct of λ*Prolog* not found in *Prolog*: embedded implication. Clause `MOD-2` attempts to prove the validity of a formula of the form $\Diamond\varphi$ by selecting a pair of unordered events, temporarily augmenting the current knowledge state with either ordering, and checking whether $\Diamond\varphi$ is valid in that extension. At worst, the process terminates when we reach a total ordering since no unordered pairs of events can be found. In case of failure, another pair of unordered events is selected. As expected from our complexity results, this trial and error strategy has an exponential cost in the worst case. We have shown in [CM99] how to alleviate the burden in specific cases. Efficient approximations are instead discussed below.

The validation of formulas of the form $\Box\varphi$ in clauses `MOD-3` and `MOD-4` reduces to the previous case by exploiting a form of double negation. A direct representation of the semantics of this operator in λ*Prolog* cannot be achieves since this language (as well as *Prolog*) lack an extensional form of universal quantification. This issue is discussed at length in [CM99].

EC

The core of the code that test whether an *EC* formula is an MVI can be found in the module `ec_base` in Appendix A.7. Clauses `EC-1` and `EC-2` provide a direct encoding of Definition 2.1, where clause `EC-2` faithfully emulates the meta-predicate *broken*. This representation is almost identical to the standard *Prolog* implementation presented in [KS86].

Notice that these clauses do not rely on the predicates `initiates`, `terminates`, and `exclusive` declared in the module `ec_structure` in Appendix A.2; this module is not even imported in `ec_base`. They instead call `init`, `term`, and `excl` defined in module `mvis`. The connection is made in module `ec` (also in Appendix A.7), which merges `ec_base` and `ec_structure`, and defines `init`, `term`, and `excl` in terms of `initiates`, `terminates`, and `exclusive` in the obvious way. Queries must be directed to this module, and not to `ec_base`.

PEC-structures and Preconditions

PEC-structures differ from *EC*-structures by the form of the initiation and termination conditions, and by the omission of the exclusivity relation, unnecessary in the presence of preconditions. This forces us to provide distinct declarations for the corresponding entities, that we have grouped in the module `pec_structure` in Appendix A.6. Specifically, given a *PEC*-structure $\mathcal{H} = (\mathbf{E},\ \mathbf{P},\ [\cdot|\cdot\rangle,\ \langle\cdot|\cdot])$, we continue to rely on the types `event` and `property` to classify the elements of $\mathbf{E}$ and $\mathbf{P}$, respectively. We instead give alternative definitions of `initiates` and `terminates` in order to accommodate contexts. Specifically, we adopt the following encoding for these entities:

$$\begin{aligned}\ulcorner[\cdot|\cdot\rangle\urcorner &= \{\texttt{initiates}\ \ulcorner e\urcorner\ulcorner p\urcorner\ulcorner C\urcorner : e\in\mathbf{E},\ p\in\mathbf{P}, C\in 2^{\mathbf{P}}, e\in[p|C\rangle\};\\ \ulcorner\langle\cdot|\cdot]\urcorner &= \{\texttt{terminates}\ \ulcorner e\urcorner\ulcorner p\urcorner\ulcorner C\urcorner : e\in\mathbf{E}, p\in\mathbf{P}, C\in 2^{\mathbf{P}}, e\in\langle p|C]\};\end{aligned}$$

where contexts are represented by using the list datatype of *λProlog*. Notice that the predicate `exclusive` is not defined.

Module `precon` in Appendix A.6 contains the code that completes module `ec_base` to obtain a faithful encoding of Definition 3.8. It is best to compare this definition with the result of merging those two modules together (which is what module `pec` does in Appendix A.9). It is then evident that the predicates `init`, `term` and `broken` to implement the meta-predicates *pInit*, *pTerm* and *pBroken* in Definition 3.8. Finally, we use the auxiliary predicate `recHolds` to iterate the MVI check over the elements of a context: clauses `PRECON-3` and `PRECON-4` emulate the context quantification appearing in the definition of *pInit* and *pTerm* as iteration over lists. Clause `PRECON-5` defines `excl` as an identity check, with the effect of making the last two lines of clause `EC-2` equivalent to

```
(init E P; term E P)
```

as prescribed in Definition 3.8. This module stands as the basis for the implementation of the calculi in the right-hand side cube in Figure 1.

MEC

The core of a polynomial implementation of *MEC* and the approximate subcalculi is contained in the module `mec_base` in Appendix A.8. It extends `ec_base` with the function symbols `must` and `may`, defined as in the case of generic modalities. The validity of modal *MEC* formulas is efficiently implemented by relying on the local conditions in Lemma 3.6. In particular, the predicate `necBroken` corresponds to the meta-predicate *necBroken* in the first part of that result.

The module `mec`, also shown in Appendix A.8, enables issuing *MEC*-queries by merging `ec` and `mec_base`.

An alternative way of implementing *MEC* is to bundle the modules `ec` and `modalities`, but proving the validity of a modal MVI may then take a time exponential in the number of recorded events.

Other Calculi

The other possible Event Calculi combinations of connectives, quantifiers, modalities, and preconditions are obtained by merging the appropriate modules defining the basic components, as specified in Figure 1. They are listed in Appendix A.9. In particular, all calculi that do not make preconditions available inherit from `ec`, while the calculi that rely on preconditions accumulate module `pec`.

Approximations

As we saw in Section 6, by applying Propositions 3.10 and 6.4, an arbitrary *QCMPEC*-formula can be rewritten into a formula where modalities enclose at most atoms of the form $p(e_1, e_2)$. This transformation, that we called *approx* in Section 6, is inexpensive and has the benefit of lowering the cost of queries to a polynomial level. However, this procedure does not maintain provability: it is neither sound nor complete in general, although either property holds for specific sublanguages. It can however serve as a useful approximation in many cases.

The λ*Prolog* implementation of this transformation as the module `approx` can be found in Appendix A.10. It relies on two predicates, "`eqcmec`" and "`approx`". The first checks whether the formula represented as its argument is an *QCMEC*-formula with the property that modalities enclose only atomic formulas. It is implemented by checking the nature of the main operator of its argument, and recursively examining its subformulas. The only clauses that deserves some discussion are the ones that traverse quantifiers. Remember that `forSomeEvent` and `forAllEvent` accept an argument of type `event -> mvi`. Therefore, when analyzing the subformula represented by their argument, we need to instantiate it with some event to obtain an object of type `mvi`. Any event would do, in this case. However, event names are chosen when encoding a specific *EC*-problem, while we would like our code to apply to arbitrary situations. We achieve the desired effect by temporarilly introducing a new event in the system by means of the universal quantification construct available in λ*Prolog* (`pi`).

The clauses defining `approx` carry the transformations expressed by Propositions 3.10 and 6.4. The first argument is the original formula, while the second argument holds a representation of the rewritten formula. The predicate `approx` operates by analyzing the structure of the source formula, usually two levels of connectives at a time. Clauses `APP-1` to `APP-6` deal with the cases where the topmost operator of the source formula is non-modal. Clauses `APP-7` to `APP-18` implement Proposition 3.10. Clauses `APP-19` to `APP-24` instead realize the approximate equivalences in Proposition 6.4. Finally, clause `APP-25` checks those formulas that are already in the target form. In this module, implications are systematically translated into negations and disjunctions, for simplicity, while quantifiers are treated as in the case of `eqcmec`.

The approximate calculi with and without preconditions are implemented in module `app_qcmec` and `app_pqcmec`, respectively. Both define the predicate `app_holds`. It validates a formula by first translating it using `approx`, and then calling `holds` on the resulting expression. Notice that `app_qcmpec` does not rely on the module `modalities`, as `mpec` does, but on `mec` and therefore transitively on `mec_base`. This allows for a modular and transparent implementation of the meta-predicates *necHolds* and *necHolds* introduced in Section 6.

7.3 Soundness and Completeness

The encoding we have chosen as an implementation of our family of event calculi permits an easy proof of its faithfulness with respect to the formal specification of this formalism. Key factors in the feasibility of this endeavor are the precise semantic definitions given in Section 3, and the exploitation of the declarative features of λ*Prolog*.

In the rest of this section, we denote with *XEC* any of the event calculi discussed in this paper. Moreover, we write $\models_{XEC}$ for the associated validity relation. Finally, we indicate with `xec` the corresponding λ*Prolog* module from Appendix A.

We first recall the following lemma from [CM99], which specifies that `before` is a sound and complete implementation of the ordering relation in the current world.

Lemma 7.1 (*Soundness and completeness of* `before` *w.r.t. transitive closure*)

Let $\mathcal{H}$ be an EC- or PEC-structure with events in E, and o a state of knowledge, then for any $e_1, e_1 \in E$

$$\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{before}\ \ulcorner e_1\urcorner\ \ulcorner e_2\urcorner \quad \textit{iff} \quad e_1 <_{o^+} e_2. \qquad \blacksquare$$

In the absence of preconditions, the soundness and completeness results for `broken` and `holds` can be conveniently staged [CM99]. In the more general setting, they dependent on each other, as well as on the similar results for `init`, `term` and `recHolds`. We have the following soundness and completeness theorem for atomic queries

Theorem 7.2 (*Soundness and completeness for atomic queries*)

Let $\mathcal{H}$ be an EC- or PEC-structure with events in E and o a state of knowledge, then

a. $\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{holds(period}\ \ulcorner e_1\urcorner\ \ulcorner p\urcorner\ \ulcorner e_2\urcorner\texttt{)}$ *iff* $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(o^+)$;

b. $\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{broken}\ \ulcorner p\urcorner\ \ulcorner e_1\urcorner\ \ulcorner e_3\urcorner$ *iff* $br(p, e_1, e_2, o^+)$ *holds in* $\mathcal{H}$;

c. $\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{init}\ \ulcorner e\urcorner\ \ulcorner p\urcorner$ *iff* $init(e, p, o^+)$ *holds in* $\mathcal{H}$;

d. $\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{term}\ \ulcorner e\urcorner\ \ulcorner p\urcorner$ *iff* $term(e, p, o^+)$ *holds in* $\mathcal{H}$;

e. $\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{recHolds}\ \ulcorner C\urcorner\ \ulcorner e\urcorner$ *iff* $\forall q \in C.\ \exists e', e'' \in E.\ q(e_1, e_2) \in \upsilon_{\mathcal{H}}(o^+)\ \wedge\ e_1 <_{o^+} e\ \wedge\ e \leq_{o^+} e_2$ *(only if $\mathcal{H}$ is a PEC-structure).*

Proof.
($\Rightarrow$) This part of the proof proceeds by simultaneous induction on the structure of the given λ*Prolog* derivations.

($\Leftarrow$) In order to proved this direction of the statement of the theorem, we must proceed by simultaneous induction on the definition of the expressions that appear on its right-hand side. More precisely, we will admit appealing to the induction hypothesis in the following circumstances:

- From a to b, c or d if the property does not change.
- From b to c, d if the property does not change.
- From c or d, to e if $\max_{q \in C}(\mathcal{B}_{\mathcal{H}}(q)) < \mathcal{B}_{\mathcal{H}}(p)$.
- From e to a if $\mathcal{B}_{\mathcal{H}}(p) \leq \max_{q \in C}(\mathcal{B}_{\mathcal{H}}(q))$.

The latter two cases are applicable only if $\mathcal{H}$ is a *PEC*-structure. It is easy to prove that, under the assumption that the dependency graph of $\mathcal{H}$ is acyclic (i.e. if $\mathcal{B}_{\mathcal{H}}$ is finite), then this specification constitutes a well-ordering, enabling us to proceed by induction. ∎

On the basis of this result, it is relatively simple to prove that the semantic characterization that enriches atomic queries with any or all of boolean connectives, quantifiers, and modalities justifies the corresponding λ*Prolog* module implementing them. This is succinctly captured by the following theorem:

Theorem 7.3 (*Soundness and Completeness*)

Let $\mathcal{H}$ be an EC- or PEC-structure with events in E, o a binary acyclic relation over E, and φ a formula in $\mathcal{L}_{\mathcal{H}}(XEC)$, then

$$\texttt{xec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{holds}\ \ulcorner\varphi\urcorner \quad \textit{iff} \quad \mathcal{I}_{\mathcal{H}}; o^+ \models_{XEC} \varphi$$ ∎

The forward direction of each instance of this theorem is proved by induction on the structure of a λ*Prolog* derivation of the sequent on the left-hand side. The base case relies on Theorem 7.2.

The backward direction results from unfolding the inductive definition of validity given in Section 3. These techniques have been rigorously deployed in [CM99] in the case of the sole *CMEC* (then called

GMEC). Therefore, we refrain from presenting a more detailed account of this simple but rather long and tedious argument. The treatment of quantifiers can instead be found in [CFM98c]. The interested reader is invited to consult those sources.

Similar results hold also in the case of the approximate calculi. Indeed, although the transformation they rely on is neither sound nor complete with respect to the source calculus of the translation, our implementation realizes it faithfully. We omit stating this property for the sake of brevity.

8 Conclusions

In this paper, we proposed an original specification framework that allows us to formally characterize the functionalities of basic *EC* and of several useful extensions. We used this formalization to define a number of event calculi that extend the range of queries accepted by *EC* by supporting advanced functionalities such as arbitrary quantification over events, modal queries, mixed connectives, and preconditions. We systematically analyzed and compared the expressiveness and complexity of the various calculi against each other. Furthermore, we provided a declarative encoding of all of these calculi in the logic programming language *λProlog* and proved the soundness and completeness of the resulting logic programs.

As for future work, we intend to use the proposed framework to formally characterize other extensions of basic *EC*, such as those adding discrete processes, time granularity, and continuous change. We are also looking for algorithms that improve the efficiency of model checking in the polynomial cases (cf. Theorem 5.1). In particular, we are working at the generalization of the graph-theoretic approach to model checking in *EC* and *MEC*, that we proposed in [FM99a, FM99b], to the other polynomial calculi. Furthermore, we are considering the issue of finding a lower bound to the complexity of the model checking problem in the tractable cases, in order to obtain a definitive yard-stick to measure the quality of the proposed polynomial model checking algorithms.

Finally, although we developed our analysis in the context of the Event Calculus, we expect it to be applicable to any formalisms for reasoning about partially ordered events. In particular, we intend to explore the applicability of the proposed approach to frameworks such as the Situation Calculus [CH69] and the Features and Fluents formalism [San94].

References

[CCM95] Iliano Cervesato, Luca Chittaro, and Angelo Montanari. A modal calculus of partially ordered events in a logic programming framework. In L. Sterling, editor, *Proceedings of the Twelfth International Conference on Logic Programming — ICLP'95*, pages 299–313, Kanagawa, Japan, 13–16 June 1995. MIT Press.

[CCM96] Iliano Cervesato, Luca Chittaro, and Angelo Montanari. A general modal framework for the event calculus and its skeptical and credulous variants. In W. Wahlster, editor, *Proceedings of the Twelfth European Conference on Artificial Intelligence — ECAI'96*, pages 33–37, Budapest, Hungary, 12–16 August 1996. John Wiley & Sons.

[CFM97a] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. A hierarchy of modal event calculi: Expressiveness and complexity. In H. Barringer, M. Fisher, D. Gabbay, , and G. Gough, editors, *Proceedings of the Second International Conference on Temporal Logic — ICTL'97*, pages 1–17, Manchester, England, 14–18 July 1997. Kluwer Applied Logic Series. Extended and revised version submitted for publication.

[CFM97b] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. Modal event calculi with preconditions. In R. Morris and L. Khatib, editors, *Proceedings of the Fourth International*

Workshop on Temporal Representation and Reasoning — TIME'97, pages 38–45, Daytona Beach, FL, 10–11 May 1997. IEEE Computer Society Press.

[CFM98a] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. The complexity of model checking in modal event calculi with quantifiers. In T. Cohn and L. Schubert, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning — KR'98*, pages 361–365, Trento, Italy, 2–5 June 1998. Morgan Kaufmann.

[CFM98b] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. The complexity of model checking in modal event calculi with quantifiers. *Journal of Electronic Transactions on Artificial Intelligence*, 2:1–23, 1998. Extended and revised version of [CFM98a].

[CFM98c] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. Event calculus with explicit quantifiers. In R. Morris and L. Khatib, editors, *Proceedings of the Fifth International Workshop on Temporal Representation and Reasoning — TIME'98*, pages 81–88, Sanibel Island, FL, 16–17 May 1998. IEEE Computer Society Press.

[CH69] John Mc Carthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[CM99] Iliano Cervesato and Angelo Montanari. A general modal framework for the event calculus and its skeptical and credulous variants. *Journal of Logic Programming*, 38(2):111–164, 1999. Extended and revised version of [CCM96].

[CMC95] Luca Chittaro, Angelo Montanari, and Iliano Cervesato. Speeding up temporal reasoning by exploiting the notion of kernel of an ordering relation. In S.D. Goodwin and H.J. Hamilton, editors, *Proceedings of the Second International Workshop on Temporal Representation and Reasoning — TIME'95*, pages 73–80, Melbourne Beach, FL, 26 April 1995.

[CMP93] Iliano Cervesato, Angelo Montanari, and Alessandro Provetti. On the non-monotonic behavior of the event calculus for deriving maximal time intervals. *International Journal on Interval Computations*, 2:83–119, 1993.

[CMP94] Luca Chittaro, Angelo Montanari, and Alessandro Provetti. Skeptical and credulous event calculi for supporting modal queries. In A. Cohn, editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence — ECAI'94*, pages 361–365. John Wiley & Sons, 1994.

[DB88] Thomas Dean and Mark Boddy. Reasoning about partially ordered events. *Artificial Intelligence*, 36:375–399, 1988.

[DMB92] Marc Denecker, Lode Missiaen, and Maurice Bruynooghe. Temporal reasoning with abductive event calculus. In B. Neumann, editor, *Proceedings of the Tenth European Conference on Artificial Intelligence — ECAI'92*, pages 384–388. John Wiley & Sons, 1992.

[Esh88] Kave Eshghi. Abductive planning with event calculus. In *Proceedings of the Fifth International Conference on Logic Programming — ICLP'88*, pages 562–579, Seattle, WA, 1988. MIT Press.

[Eva90] Chris Evans. The macro-event calculus: Representing temporal granularity. In *Proceedings of the Pacific Rim International Conference on Artificial Intelligence — PRICAI'90*, Nagoya, Japan, 1990. IOS Press.

[FM99a] Massimo Franceschet and Angelo Montanari. A graph-theoretic approach to efficiently reasoning about partially ordered events in the event calculus. In C. Dixon and M. Fisher, editors, *Proceedings of the 6th International Workshop on Temporal Representation and Reasoning — TIME'99*, pages 55–66. IEEE Computer Society Press, 1999.

[FM99b] Massimo Franceschet and Angelo Montanari. Pairing transitive closure and reduction to efficiently reason about partially ordered events. In M. Thielscher, editor, *Proceedings of the 3rd International Workshop on Nonmonotonic Reasoning, Action and Change at IJCAI99 — NRAC'99*, pages 79–86, 1999.

[KS86] Robert Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[KW90] Elpida Keravnou and John Washbrook. A temporal reasoning framework used in the diagnosis of skeletal dysplasias. *Artificial Intelligence in Medicine*, 2:239–265, 1990.

[Mil96] Dale Miller. Lambda Prolog: An introduction to the language and its logic. Current draft available from `http://cse.psu.edu/~dale/lProlog`, 1996.

[MMCR92] Angelo Montanari, Enrico Maim, Emanuele Ciapessoni, and Elena Ratto. Dealing with time granularity in the event calculus. In *Proceedings of the International Conference on Fifth Generation Computer Systems — FGCS'92*, pages 702–712, Tokyo, Japan, 1–5 June 1992.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Nök91] Klaus Nökel. *Temporarilly Distributed Symptoms in Technical Diagnosis*. Springer-Verlag, 1991.

[Pap94] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[San94] Erik Sandewall. *Features and Fluents: A Systematic Approach to theh Representation of Knowledge about Dynamical Systems*. Oxford University Press, 1994.

[Sch95] Dirk Schroeder. *Staying Healthy in Asia, Africa and Latin America*. Moon publications, 1995.

[Sha89] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence — IJCAI'89*, pages 1050–1055, Detroit, MI, 1989. Morgan Kauffman.

[Sha90] Murray Shanahan. Representation of continuous change in the event calculus. In *Proceedings of the Nineth Conference on Artificial Intelligence — ECAI'90*, pages 598–603, Stockholm, Sweden, 1990. John Wiley & Sons.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1994.

A Code

A.1 Orderings

```
module ordering.

kind event       type.

type beforeFact  event -> event -> o.
type before      event -> event -> o.

before E1 E2 :-                                  % Ord-1 %
        beforeFact E1 E2.
before E1 E2 :-                                  % Ord-2 %
        beforeFact E1 E, before E E2.
```

A.2 *EC*-structures and MVIs

```
module ec_structure.

kind event      type.
kind property   type.

type initiates  event -> property -> o.
type terminates event -> property -> o.
type exclusive  property -> property -> o.
type happens    event -> o.


module mvis.

kind event      type.
kind property   type.

type init       event -> property -> o.
type term       event -> property -> o.
type excl       property -> property -> o.
type happens    event -> o.

kind mvi        type.
type period     event -> property -> event -> mvi.

type holds      mvi -> o.
```

A.3 Connectives

```
module connectives.
accumulate mvis.

type neg      mvi -> mvi.
type and      mvi -> mvi -> mvi.      infixr and     5.
type or       mvi -> mvi -> mvi.      infixr or      5.
type implies  mvi -> mvi -> mvi.      infixl implies 4.

holds (neg X)       :-  not (holds X).        % CONN-1 %
holds (X and Y)     :-  holds X, holds Y.     % CONN-2 %
holds (X or Y)      :-  holds X; holds Y.     % CONN-3 %
holds (X implies Y) :-  holds ((neg X) or Y). % CONN-4 %
```

A.4 Quantifiers

```
module quantifiers.
accumulate mvis.

type forAllEvent  (event -> mvi) -> mvi.
type forSomeEvent (event -> mvi) -> mvi.

holds (forAllEvent X) :-                        % QUANT-1 %
        not (sigma E \ (happens E,
                        not (holds (X E)))).
holds (forSomeEvent X) :-                       % QUANT-2 %
sigma E \ holds (X E).
```

A.5 Generic Modalities

```
module modalities.
accumulate mvis, ordering.

type must     mvi -> mvi.
type may      mvi -> mvi.

% ------- May-formulas
holds (may X) :-                               % MOD-1 %
        holds X.

holds (may X) :-                               % MOD-2 %
        happens E1, happens E2,
        not (E1 = E2),
        not (before E1 E2),
        not (before E2 E1),
        beforeFact E1 E2 =>
             holds (may X).

% ------- Must-formulas
type auxMust  mvi -> o.

holds (must X) :-                              % MOD-3 %
        holds X,
        not (auxMust X).

auxMust X :-                                   % MOD-4 %
        happens E1, happens E2,
        not (E1 = E2),
        not (before E1 E2),
        not (before E2 E1),
        beforeFact E1 E2 =>
             not (holds (must X)).
```

A.6 *PEC*-structures and Preconditions

```
module pec_structure.

kind event      type.
kind property   type.

type initiates  event -> property -> list property -> o.
type terminates event -> property -> list property -> o.
type happens    event -> o.


module precon.
accumulate ordering, mvis, pec_structure.

type recHolds event -> list property  -> o.

init E P :-                                  % PRECON-1 %
        initiates E P C,
        recHolds E C.

term E P :-                                  % PRECON-2 %
        terminates E P C,
        recHolds E C.

recHolds E (Q :: C) :-                       % PRECON-3 %
        holds (period E' Q E''),
        before E' E,
        (before E E''; E = E'').
recHolds E nil.                              % PRECON-4 %

excl P P.                                    % PRECON-5 %
```

A.7 *EC*

```
module ec_base.
accumulate mvis, ordering.
```

```
type broken   event -> property -> event -> o.

holds (period Ei P Et) :-                  % EC-1 %
        happens Ei, init Ei P,
        happens Et, term Et P,
        before Ei Et, not (broken Ei P Et).
broken Ei P Et :-                          % EC-2 %
        happens E,
        before Ei E, before E Et,
        (P = Q; excl P Q),
        (init E Q; term E Q).

module ec.
accumulate ec_base, ec_structure.

init E P :- initiates E P.                 % ECAUX-1 %
term E P :- terminates E P.                % ECAUX-2 %
excl P Q :- exclusive P Q.                 % ECAUX-3 %
excl P Q :- exclusive Q P.                 % ECAUX-4 %
```

A.8 *MEC*

```
module mec_base.
accumulate ec_base.

type must            mvi -> mvi.
type may             mvi -> mvi.
type necBroken       event -> property -> event -> o.


holds (must (period Ei P Et)) :-           % MEC-1 %
        happens Ei, init Ei P,
        happens Et, term Et P,
        before Ei Et,
        not (necBroken Ei P Et).

necBroken Ei P Et :-                       % MEC-2 %
        happens E,
        not (E = Ei), not (E = Et),
        not (before E Ei),
        not (before Et E),
        (init E Q; term E Q),
        (excl P Q; P = Q).

holds (may (period Ei P Et)) :-            % MEC-3 %
        happens Ei, init Ei P,
        happens Et, term Et P,
        not (before Et Ei),
        not (broken Ei P Et).

module mec.
accumulate ec, mec_base.
```

A.9 Other Calculi

```
module cec.
accumulate ec, connectives.

module qec.
accumulate ec, quantifiers.

module pec.
accumulate ec_base, precon.

module cmec.
accumulate ec, connectives, modalities.

module qmec.
accumulate ec, modalities, quantifiers.

module mpec.
accumulate pec, modalities.

module qcec.
accumulate ec, connectives, quantifiers.

module cpec.
accumulate pec, connectives.

module qpec.
accumulate pec, quantifiers.

module qcmec.
accumulate ec, connectives, modalities, quantifiers.

module cmpec.
accumulate pec, connectives, modalities.

module qmpec.
accumulate pec, modalities, quantifiers.

module qcpec.
accumulate pec, connectives, quantifiers.

module qcmpec.
accumulate pec, connectives, modalities, quantifiers.
```

A.10 Approximations

```
module approx.
accumulate connectives, quantifiers.

type must      mvi -> mvi.
type may       mvi -> mvi.

type approx    mvi -> mvi -> o.
type eqcmec    mvi -> o.

% Non-modal formulas

approx (neg X) (neg Z) :-                  % APP-1 %
        approx X Z.

approx (X1 and X2) (Z1 and Z2) :-          % APP-2 %
        approx X1 Z1,
        approx X2 Z2.

approx (X1 or X2) (Z1 or Z2) :-            % APP-3 %
        approx X1 Z1,
        approx X2 Z2.

approx (X1 implies X2) Z) :-               % APP-4 %
        approx ((neg X1) or X2) Z.

approx (forAllEvent E \ (X E))
       (forAllEvent E \ (Z E)) :-          % APP-5 %
        pi e \ approx (X e) (Z e).

approx (forSomeEvent E \ (X E))
       (forSomeEvent E \ (Z E)) :-         % APP-6 %
        pi e \ approx (X e) (Z e).

% Equivalences

approx (must (neg X)) (neg Z) :-           % APP-7 %
        approx (may X) Z.

approx (may (neg X)) (neg Z) :-            % APP-8 %
        approx (must X) Z.

approx (must (X1 and X1)) (Z1 and Z2) :-   % APP-9 %
        approx (must X1) Z1,
        approx (must X2) Z2.

approx (may (X1 or X1)) (Z1 or Z2) :-      % APP-10 %
```

```
        approx (may X1) Z1,
        approx (may X2) Z2.

approx (must (X1 implies X2)) Z :-          % APP-11 %
        approx (must ((neg X1) or X2)) Z.

approx (may (X1 implies X2)) Z :-           % APP-12 %
        approx (may ((neg X1) or X2)) Z.

approx (must (forAllEvent E \ (X E)))
         (forAllEvent E \ (Z E)) :-         % APP-13 %
        pi e \ approx (must (X e)) (Z e).

approx (may (forSomeEvent E \ (X E)))
         (forSomeEvent E \ (Z E)) :-        % APP-14 %
        pi e \ approx (may (X e)) (Z e).

approx (must (must X)) Z :-                 % APP-15 %
        approx (must X) Z.

approx (may (may X)) Z :-                   % APP-16 %
        approx (may X) Z.

approx (must (may X)) Z :-                  % APP-17 %
        approx (may X) Y,
        approx (must Y) Z.

approx (may (must X)) Z :-                  % APP-18 %
        approx (must X) Y,
        approx (may Y) Z.

% Complete but unsound approximations

approx (may (X1 and Y1)) (Z1 and Z2) :-     % APP-19 %
        approx (may X1) Z1,
        approx (may X2) Z2.

approx (may (forAllEvent E \ (X E)))
         (forAllEvent E \ (Z E)) :-         % APP-20 %
        pi e \ approx (may (X e)) (Z e).

approx (may (must (period Ei P Et)))
          (may (period Ei P Et)).           % APP-21 %


% Sound but incomplete approximations

approx (must (X1 or Y1)) (Z1 or Z2) :-      % APP-22 %
        approx (must X1) Z1,
        approx (must X2) Z2.

approx (must (forSomeEvent E \ (X E)))
          (forSomeEvent E \ (Z E)) :-       % APP-23 %
        pi e \ approx (must (X e)) (Z e).

approx (must (may (period Ei P Et)))
         (must (period Ei P Et)).           % APP-24 %

% Base case

approx X X :- eqcmec X.                     % APP-25 %


% E-QCMEC formulas

eqcmec (period Ei P Et).                    % EQCMEC-1 %

eqcmec (must (period Ei P Et)).             % EQCMEC-2 %

eqcmec (may (period Ei P Et)).              % EQCMEC-3 %

eqcmec (neg X) :- eqcmec X.                 % EQCMEC-4 %

eqcmec (X1 and X2) :-                       % EQCMEC-5 %
        eqcmec X1,
        eqcmec X2.

eqcmec (X1 or X2) :-                        % EQCMEC-6 %
        eqcmec X1,
        eqcmec X2.

eqcmec (X1 implies X2) :-                   % EQCMEC-7 %
        eqcmec ((neg X1) or X2.

eqcmec (forAllEvent E \ (X E)) :-
        pi e \ eqcmec (X e).                % EQCMEC-8 %

eqcmec (forSomeEvent E \ (X E)) :-          % EQCMEC-9 %
        pi e \ eqcmec (X e).

module app_qcmec.
accumulate approx, qcmec.

type app_holds  mvi -> o.

app_holds X :-                              % AQCM %
        approx X Z,
        holds Z.


module app_qcmpec.
accumulate approx, mec, connectives, quantifiers, precon.

type app_holds  mvi -> o.

app_holds X :-                              % APQCM %
        approx X Z,
        holds Z.
```

B Case Studies

B.1 Diagnosing Faulty Hardware

```
module cncc.
accumulate cmpec.

type test_plain o.
type test_must  o.
type test_may   o.

type zero29 property.
type one29  property.
type zero30 property.
type one30  property.

type e1 event.   happens e1.
type e2 event.   happens e2.
type e3 event.   happens e3.
type e4 event.   happens e4.

initiates  e1 zero29 nil.
initiates  e2 zero30 (zero29 :: nil).
initiates  e3 one30  nil.
initiates  e4 one29  nil.

terminates e1 one29  nil.
terminates e2 one30  nil.
terminates e3 zero30 (zero29 :: nil).
terminates e4 zero29 nil.

% Original state
beforefact e1 e3.   beforefact e1 e4.
beforefact e2 e3.   beforefact e2 e4.

% Expected completion
%beforefact e1 e2.
%beforefact e3 e4.

% Alternative extension
%beforefact e2 e1.
```

```
test_plain :- holds       (period e2 zero30 e3).
test_must  :- holds (must (period e2 zero30 e3)).
test_may   :- holds (may  (period e2 zero30 e3)).
```

B.2 Diagnosing Metatropic Dwarfism

```
module dwarfism.
accumulate cmpec.

type test_plain o.
type test_must o.
type test_may o.

type narrow_th      property.
type normal_th      property.
type wide_th        property.
type mild_sc        property.
type moderate_sc    property.
type progressive_sc property.

type e0 event.   happens e0.
type e1 event.   happens e1.
type e2 event.   happens e2.
type e3 event.   happens e3.
type e4 event.   happens e4.
type e5 event.   happens e5.
type e6 event.   happens e6.

initiates  e0 narrow_th      nil.
initiates  e1 mild_sc        nil.
initiates  e2 moderate_sc    nil.
initiates  e3 normal_th      (moderate_sc :: nil).
initiates  e4 progressive_sc nil.
initiates  e5 wide_th        (progressive_sc :: nil).

terminates e2 mild_sc        nil.
terminates e3 narrow_th      nil.
terminates e4 moderate_sc    nil.
terminates e5 normal_th      nil.
terminates e6 wide_th        nil.
terminates e6 progressive_sc nil.

% Original state
beforeFact e0 e1.   beforeFact e1 e2.
beforeFact e1 e3.   beforeFact e2 e4.
beforeFact e2 e5.   beforeFact e3 e4.
beforeFact e3 e5.   beforeFact e4 e6.
beforeFact e5 e6.

% Expected completion
%beforeFact e2 e3.
%beforeFact e4 e5.

% Alternative extension
%beforeFact e3 e2.

test_plain :-
        holds (          (period e3 normal_th e5)
                     and (period e5 wide_th   e6)).
test_must  :-
        holds (must (    (period e3 normal_th e5)
                     and (period e5 wide_th   e6))).
test_may   :-
        holds (may  (    (period e3 normal_th e5)
                     and (period e5 wide_th   e6))).
```

B.3 Diagnosing Malaria

```
module malaria.
accumulate qcec.

% Ordering
type precedes  event -> event -> mvi.  infixr precedes 6.
holds (E1 precedes E2) :-  before E1 E2.     % PREC %

type fever   property.      prop fever.
type chills  property.      prop chills.
type malaria o.

malaria :- holds (forAllEvent E1 \
                  forAllEvent E2 \
                    ((period E1 chills E2) implies
                        (forSomeEvent E1' \
                         forSomeEvent E2' \
                           ((E1 precedes E1') and
                            (E1' precedes E2) and
                            (period E1' fever E2'))))).

type  e1 event.   happens  e1.   initiates   e1 chills.
type  e2 event.   happens  e2.   initiates   e2 fever.
type  e3 event.   happens  e3.   terminates  e3 chills.
type  e4 event.   happens  e4.   terminates  e4 fever.
type  e5 event.   happens  e5.   initiates   e5 chills.
type  e6 event.   happens  e6.   initiates   e6 fever.
type  e7 event.   happens  e7.   terminates  e7 chills.
type  e8 event.   happens  e8.   terminates  e8 fever.
type  e9 event.   happens  e9.   initiates   e9 chills.
type e10 event.   happens e10.   initiates  e10 fever.
type e11 event.   happens e11.   terminates e11 chills.
type e12 event.   happens e12.   terminates e12 fever.

beforeFact  e1  e2.    beforeFact  e2  e3.
beforeFact  e3  e4.    beforeFact  e4  e5.
beforeFact  e5  e6.    beforeFact  e6  e7.
beforeFact  e7  e8.    beforeFact  e8  e9.
beforeFact  e9 e10.    beforeFact  e10 e11.
beforeFact  e11 e12.
```