1 Introduction

The *Event Calculus*, abbreviated EC [9], 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. 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.

A systematic analysis of EC has recently been undertaken in order to gain a better understanding of this calculus and determine ways of augmenting its expressive power. The keystone of this endeavor has been the definition of an extendible formal specification of the functionalities of this formalism [3]. This has had the effects of establishing a semantic reference against which to verify the correctness of implementations [6], of casting EC as a model checking problem [4], and of setting the ground for studying the complexity of this problem, which was proved polynomial [2]. Extensions of this model have been designed to accommodate constructs intended to enhance the expressiveness of EC. In particular, modal versions of EC [1], the interaction between modalities and connectives [4], and preconditions [5] have all been investigated in this context.

In this paper, we continue this endeavor to enhance the expressive power of EC by considering the possibility of quantifying over events in queries, in conjunction with boolean connectives and modal operators. We also admit requests to check the relative order of two events. We thoroughly analyze the representational and computational features of the resulting formalism, that we call *QCMEC*. We also consider two proper sublanguages of it, *EQCMEC*, in which modalities are applied to atomic formulas only, and *CMEC*, which is quantifier-free. We show that *QCMEC* and its restrictions can effectively be used to encode diagnosis problems. Moreover, we provide an elegant implementation in the higher-order logic programming language λ*Prolog* [10] and prove its soundness and completeness. As far as computational complexity is concerned, we prove that model checking in *CMEC*, *EQCMEC*, and *QCMEC* is **PSPACE**-complete. However, while solving an *EQCMEC* problem is exponential in the size of the query, it has only polynomial cost in the number n of events, thus making *EQCMEC* a viable formalism for *MVI* verification or computation. Since in most realistic applications the size of databases (n) dominates by several orders of magnitude the size of the query, n is asymptotically the parameter of interest.

The main contributions of this work are: (1) the extension of a family of modal event calculi with quantifiers; (2) permitting queries to mention ordering information; (3) the use of the higher-order features of modern logic programming languages in temporal reasoning; and (4) analyzing the complexity of model checking in these extensions of EC.

This paper is organized as follows. In Section 2, we formalize *QCMEC* and significant subcalculi. Section 3 exemplifies how this calculus can adequately model certain hardware diagnosis problems. In Section 4, we briefly introduce the logic programming language λ*Prolog*, give an implementation of *QCMEC* in it and prove the soundness and completeness of the resulting program. We study the complexity of *QCMEC* and its sublanguages in Section 5. We outline directions of future work in Section 6.

2 Modal Event Calculi with Quantifiers

In this section, we first briefly recall the syntax and semantics of a number of modal event calculi. We invite the interested reader to consult [1, 3, 4, 8, 9] for motivations, examples, properties, and technical details. We then extend these basic definitions to give a semantic foundation to refinements of these calculi with quantifiers.

2.1 Event Calculus

The Event Calculus (EC) [9] and the extensions we propose aim at modeling scenarios 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. We formalize the time-independent aspects of a situation by means of an *EC-structure* [1], defined as follows:

Definition 2.1 (*EC-structure*)

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

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

As in the original EC paper [9], we define the initiating and terminating maps in terms of event occurrences rather than event types. The latter approach can however easily be accomodate in our setting.

The temporal aspect of EC is given by the order in which events happen. Unlike the original presentation [9], we focus our attention on situations where the occurrence time of events is unknown and only assume the availability of incomplete information about the relative order in which they have happened. We however require the temporal data to be consistent so that an event cannot both precede and follow some other event. Therefore, we formalize the time-dependent aspect of a scenario modeled by EC by means of a (strict) *partial order*, i.e. an irreflexive and transitive relation, over the involved set of event occurrences. We write $W_{\mathcal{H}}$ for the set of all partial orders over the set of events E in an EC-structure $\mathcal{H}$, use the letter w to denote individual orderings, or *knowledge states*, and write $e_1 <_w e_2$ to indicate that e_1 precedes e_2 in w. 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. An *extension* of a knowledge state w is any element of $W_{\mathcal{H}}$ that contains w as a subset. We write $\mathrm{Ext}_{\mathcal{H}}(w)$ for the set of all extensions of the ordering w in $W_{\mathcal{H}}$.

Given a structure $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ and a knowledge state w, EC permits inferring the *maximal validity intervals*, or *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 holds maximally. Consequently, we adopt as the *query language* of EC the set $\mathcal{L}_{\mathcal{H}}(EC) = \{p(e_1, e_2) : p \in P \text{ and } e_1, e_2 \in E\}$ of all such property-labeled intervals over $\mathcal{H}$. We interpret the elements of $\mathcal{L}_{\mathcal{H}}(EC)$ as propositional letters and the task performed by EC reduces to deciding which of these formulas are MVIs in the current knowledge state w and which are not. This is a model checking problem.

In order for $p(e_1, e_2)$ to be an MVI relative to the event ordering w, it must be the case that $e_1 <_w e_2$. 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. The maximality requirement is caught by the negation of the meta-predicate $br(p, e_1, e_2, w)$ below, which expresses the fact that the validity 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 validity of $p(e_1, e_2)$.

Definition 2.2 (*Intended model of EC*)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be a EC-structure. The* intended *EC-*model *of* $\mathcal{H}$ *is the propositional valuation* $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \rightarrow 2^{\mathcal{L}_{\mathcal{H}}(EC)}$*, where* $p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w)$ *if and only if* (i) $e_1 <_w e_2$, (ii) $e_1 \in [p\rangle$, (iii) $e_2 \in \langle p]$, (iv) $br(p, e_1, e_2, w)$ *does not hold, where* $br(p, e_1, e_2, w)$ *abbreviates*

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

Definition 2.2 states that an event e interrupts the validity of a property if it initiates or terminates p itself or a property q which is incompatible with p. As pointed out in [6], this rule adopts the so-called *strong interpretation* of the initiate and terminate relations: a given property p ceases to hold over a time interval if an event e which initiates or terminates p, or a property incompatible with p, occurs during this interval. The strong interpretation is needed when dealing with incomplete sequences of events or, as in our case, incomplete information about their ordering. For example, consider a switch that can take two different positions: on and off. Its behavior can be described by means of two types of event: one that changes the position from off to on (turn-on), the other from on to off (turn-off). While two turn-on events cannot occur consecutively in the real world, it may happen that an incomplete sequence consisting of two consecutive turn-on events, followed by a turn-off event, is recorded in the database. The strong interpretation of the initiate relation allows EC to recognize that a missing turn-off event must have occurred between the two turn-on events. However, since it is not possible to temporally locate such an event, EC only concludes that the switch is on between the second turn-on event and the turn-off event, and it considers the first turn-on event as a pending initiating event. It is worth mentioning that an alternative interpretation of the initiate and terminate relations, called *weak interpretation*, is also possible. According to this interpretation, a property p is triggered by an initiating event unless it has already been started and not yet terminated. The weak interpretation is needed to aggregate homogeneous states. Further details about the strong/weak distinction, including its formalizations and other examples, can be found in [6].

2.2 Modal EC with Connectives

The query language of the basic EC we just formalized suffers from a remarkably low expressive power that prevents its use for modelling any but the most trivial applications. The expressiveness of this formalism is dramatically augmented by admitting boolean connectives in queries. This allows inquiring about logical combinations of basic MVI verification problems.

In our specific setting, where the ordering of event occurrences is only partially specified, the set of MVIs computed by EC is not stable with respect to the acquisition of new ordering information. Indeed, as we move to an extension of the current knowledge state, some MVIs might become invalid and new MVIs can emerge [7]. Extending the query language of EC with the modal logic operators $\Box$ and $\Diamond$ leads to the possibility of 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 [1, 8]. Several ways of combining boolean connectives and modalities, with different cost and expressiveness, have been proposed [3, 4].

In this paper, we also include a *precedence test* operator, written $<$, which allows checking the relative order of two events in the current knowledge state. In previous work, this was awkwardly achieved either by augmenting EC-structures with dedicated properties [4], or by using preconditions [5]. A native precedence test makes inquiring about the relative order of two events independent from the underlying EC-structure.

Given an EC-structure $\mathcal{H}$, the query language that freely includes these three extensions is formally defined by the following grammar:

$$\begin{array}{rcl}\varphi & ::= & p(e_1, e_2) \mid e_1 < e_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \\ & & \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi.\end{array}$$

We call this language $\mathcal{L}_{\mathcal{H}}(CMEC)$ and $CMEC$ the relative extension of EC. In addition to the above operators, we admit implication as a derived connective, where $\varphi_1 \supset \varphi_2$ is classically defined as $\neg\varphi_1 \vee \varphi_2$.

In order to formalize the semantics of the modalities in $CMEC$, we must shift the focus from the current knowledge state w to all knowledge states that are reachable from w, i.e. $\mathrm{Ext}_{\mathcal{H}}(w)$. Since $\subseteq$ is a reflexive partial order, $(W_{\mathcal{H}}, \subseteq)$ can be naturally viewed as a finite, reflexive, transitive and antisymmetric modal frame. If we consider this frame together with the straightforward modal extension of the valuation $\upsilon_{\mathcal{H}}$ to an arbitrary knowledge state, we obtain a modal model for $CMEC$. Connectives are handled as usual and incorporating the precedence test is trivial.

Definition 2.3 (*Intended model of CMEC*)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure. The* intended CMEC-model *of* $\mathcal{H}$ *is the modal model* $\mathcal{I}_{\mathcal{H}} = (W_{\mathcal{H}}, \subseteq, \upsilon_{\mathcal{H}})$*, where the propositional valuation* $\upsilon_{\mathcal{H}} : W_{\mathcal{H}} \to 2^{\mathcal{L}_{\mathcal{H}}(EC)}$ *is defined as in Definition 2.2. Given* $w \in W_{\mathcal{H}}$ *and* $\varphi \in \mathcal{L}_{\mathcal{H}}(CMEC)$*, 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:*

$$\begin{array}{lll}
\mathcal{I}_{\mathcal{H}}; w \models p(e_1, e_2) & \textit{iff} & p(e_1, e_2) \in \upsilon_{\mathcal{H}}(w); \\
\mathcal{I}_{\mathcal{H}}; w \models e_1 < e_2 & \textit{iff} & e_1 <_w e_2; \\
\mathcal{I}_{\mathcal{H}}; w \models \neg\varphi & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \not\models \varphi; \\
\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \ \textit{ and } \ \mathcal{I}_{\mathcal{H}}; w \models \varphi_2; \\
\mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \vee \varphi_2 & \textit{iff} & \mathcal{I}_{\mathcal{H}}; w \models \varphi_1 \ \textit{ or } \ \mathcal{I}_{\mathcal{H}}; w \models \varphi_2; \\
\mathcal{I}_{\mathcal{H}}; w \models \Box\varphi & \textit{iff} & \textit{for all } w' \in \mathrm{Ext}_{\mathcal{H}}(w),\ \mathcal{I}_{\mathcal{H}}; w' \models \varphi; \\
\mathcal{I}_{\mathcal{H}}; w \models \Diamond\varphi & \textit{iff} & \textit{there is } w' \in \mathrm{Ext}_{\mathcal{H}}(w) \textit{ s. t. } \mathcal{I}_{\mathcal{H}}; w' \models \varphi. \quad \Box
\end{array}$$

Notice that deciding the truth of a modal formula requires the exploration of all the extensions of the current knowledge state. Since there are exponentially many, this raises the complexity of *CMEC* beyond tractability [4]. This distressing fact is overcome in the calculus *ECMEC* [6, 4], that restricts *CMEC* by allowing $\Box$ and $\Diamond$ to enclose only atomic formulas of the form $e_1 < e_2$ and $p(e_1, e_2)$. To determine the truth of atomic formulas prefixed by one modal operator, it is possible to exploit necessary and sufficient *local conditions* over the given partial order, thus avoiding a complete (and expensive) search of all the consistent extensions of the given order [4]. Therefore, solving modal queries in *ECMEC* has polynomial cost [4].

This is particularly appealing since numerous *CMEC*-formulas are logically equivalent to *ECMEC*-formulas. The transformation proceeds by pushing the modalities inside the scope of the connectives. An *ECMEC* formula cannot always be produced since $\Box$ does not distribute over $\vee$, and dually $\Diamond$ cannot be pushed inside a conjunction. We will now consider conditions that permit overcoming this difficulty in situations of interest.

Specifically, we consider EC-structures $\mathcal{H} = (E,\ P, [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ where every property is initiated and terminated by at most one event and there are no exclusive properties. We call this condition the *singleton condition*. An atomic formula $p(e_1, e_2)$ on $\mathcal{H}$ is an MVI relative to the knowledge state $w \in W_{\mathcal{H}}$ if and only if e_1 initiates p, e_2 terminate p and (e_1, e_2) belongs to w. Indeed, the singleton condition ensures us that there are no interrupting events for p in (e_1, e_2) and thus we do not need to check whether $br(e_1, p, e_2, w)$ holds since this meta-predicate will be trivially false. The singleton condition offers further opportunities to push modalities inside the scope of connectives. We omit the proof of the following simple proposition.

Proposition 2.4 (*Consequences of the singleton condition*)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure that satisfies the singleton condition. Let* φ *be a* CMEC*-formula. For* $p \in P$ *and* $e_1, e_2 \in E$*, let* $\nu_p(e_1, e_2)$ *be either* $p(e_1, e_2)$ *or* $(e_1 < e_2)$*. Then, for any* $w \in W_{\mathcal{H}}$ *such that* $e_1 <_w e_2$*, we have that:*

i. $w \models \Box(\nu_p(e_1, e_2) \vee \varphi)$ *iff* $w \models \nu_p(e_1, e_2) \vee \Box\varphi$;
ii. $w \models \Diamond(\nu_p(e_1, e_2) \wedge \varphi)$ *iff* $w \models \nu_p(e_1, e_2) \wedge \Diamond\varphi$. ■

In particular, for $\varphi = \mathbf{false}$ (resp. $\mathbf{true}$), we have that $w \models \Box\nu_p(e_1, e_2)$ (resp. $w \models \Diamond\nu_p(e_1, e_2)$) *iff* $w \models \nu_p(e_1, e_2)$.

2.3 Modal EC with Connectives and Quantifiers

We will now enrich *CMEC* with explicit universal and existential event quantifiers that can be used freely in a query. We call the resulting formalism *QCMEC*. A logic programming implementation of *CMEC* can emulate only restricted forms of existential quantification by means of unification, while universally quantified queries are out of reach.

In order to accommodate quantifiers, we extend the query language of an *EC*-structure $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ in several respects. We first assume the existence of infinitely many *event variables* that we denote x, possibly subscripted. We write $\bar{e}$ for a syntactic entity that is either an event in E or an event variable. The query language of *QCMEC*, denoted $\mathcal{L}_{\mathcal{H}}(QCMEC)$, is the set of *closed* formulas generated by the following grammar:

$$\begin{aligned} \varphi \ ::= \ & p(\bar{e}_1, \bar{e}_2) \mid \bar{e}_1 < \bar{e}_2 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \\ & \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \forall x.\,\varphi \mid \exists x.\,\varphi. \end{aligned}$$

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 $e \in E$ 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 notion of intended model to accommodate quantifiers.

Definition 2.5 (*Intended model of QCMEC*)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure. The* intended *QCMEC*-model *of* $\mathcal{H}$ *is the modal model* $\mathcal{I}_{\mathcal{H}} = (W_{\mathcal{H}}, \subseteq, \upsilon_{\mathcal{H}})$ *defined as in Definition 2.3. Given* $w \in W_{\mathcal{H}}$ *and a (closed) formula* $\varphi \in \mathcal{L}_{\mathcal{H}}(QCMEC)$*, the truth of* φ *at* w *with respect to* $\mathcal{I}_{\mathcal{H}}$*, denoted as* $\mathcal{I}_{\mathcal{H}}; w \models \varphi$*, is defined as in Definition 2.3 with the addition of the following two cases:*

$\mathcal{I}_{\mathcal{H}}; w \models \forall x.\, \varphi$ *iff for all* $e \in E$, $\mathcal{I}_{\mathcal{H}}; w \models [e/x]\varphi$;
$\mathcal{I}_{\mathcal{H}}; w \models \exists x.\, \varphi$ *iff there exists* $e \in E$ *s. t.* $\mathcal{I}_{\mathcal{H}}; w \models [e/x]\varphi$. □

The well-foundedness of this definition derives from the observation that if $\forall x.\, \varphi$ and $\exists x.\, \varphi$ are closed formula, so is $[e/x]\varphi$ for every event $e \in E$.

A universal quantification over a finite domain can always be expanded into a finite sequence of conjunctions. Similarly an existentially quantified formula is equivalent to the disjunction of all its instances. The following lemma, whose simple proof we omit, applies these principles to *QCMEC*.

Lemma 2.6 (*Unfolding quantifiers*)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC*-structure, *with* $E = \{e_1, \ldots, e_n\}$*. Then, for every* $w \in W_{\mathcal{H}}$*,*

i. $\mathcal{I}_{\mathcal{H}}; w \models \forall x.\, \varphi$ *iff* $\mathcal{I}_{\mathcal{H}}; w \models \bigwedge_{i=1}^{n} [e_i/x]\varphi$;
ii. $\mathcal{I}_{\mathcal{H}}; w \models \exists x.\, \varphi$ *iff* $\mathcal{I}_{\mathcal{H}}; w \models \bigvee_{i=1}^{n} [e_i/x]\varphi$. ■

This property hints at the possibility of compiling a *QCMEC* query to a quantifier-free formula. Observe however that this is possible only after an *EC*-structure has been specified. We will rely on the above lemma in order to analyze the explicit complexity of the formalism in Section 5. It is also possible to take advantage of it in order to structure an implementation of *QCMEC* into a preprocessor that expands quantifiers into exhaustive sets of conjunctions or disjunctions, and a *CMEC* checker that verifies the resulting formula. We will however follow a more direct approach in Section 4.

We conclude this section by defining a quantified variant of the previously introduced formalism *ECMEC*. The calculus *EQCMEC* differs from *QCMEC* by imposing that propositional connectives and quantifiers be external to the scope of the modal operators.

3 Example

In this section, we consider a case study taken from the domain of hardware fault diagnosis that shows how an extension of *EC* with quantifiers, connectives and modalities can be conveniently used to model real-world applications.

We focus our attention on the representation and information processing of fault symptoms that are spread over periods of time and for which current expert system technology is particularly deficient [11]. Consider the following example, which diagnoses a fault in a computerized numerical control center for a production chain.

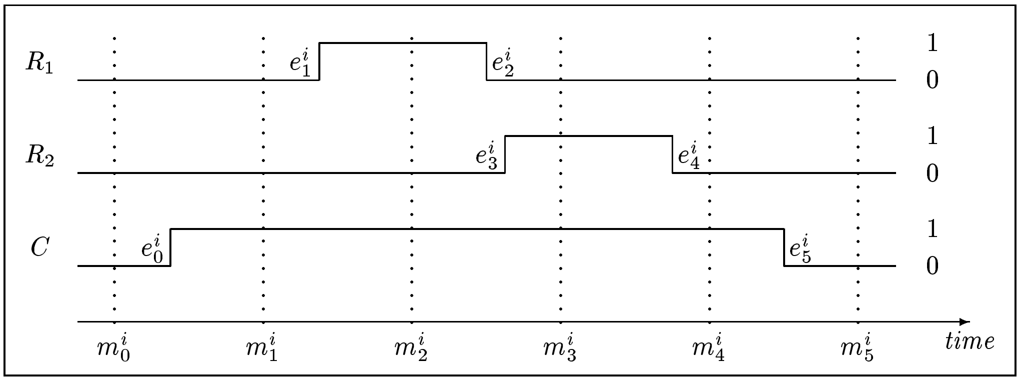

Figure 1: Expected Register Behavior and Measurements during Session i.

> *A possible cause for an undefined position of the tool magazine is a faulty limit switch S. This cause can however be ruled out if the status registers R_1 and R_2 show the following behavior in every session: from a situation in which both registers contain the value 0, they assume the value 1 in successive and disjoint time intervals (first R_1 and then R_2), and then return to 0. A session is a time interval initiated when a special register C is set to 1 and terminated when C is reset to 0.*

Figure 1 describes a possible sequence of transitions for R_1 and R_2 within an individual session i. If every recorded session has a similar pattern, the eventuality of S being faulty can be excluded. In order to verify this behavior, the contents of the registers must be monitored over time. Typically, each value (0 or 1) of a register persists for at least t time units. Measurements are made at fixed intervals (sampling periods), asynchronously with the change of value of the registers. In order to avoid losing register transitions, measurements must be made frequently enough, that is, the sampling period must be less than t. However, it may happen that transitions of *different* registers take place between two consecutive measurements, making it impossible to recover their relative order.

This situation is depicted in Figure 1, where dotted lines indicate measurements. Moreover, we have given names to the individual transitions of state of the different registers. In this specific situation, the values found at measurements m_0^i, m_1^i and m_2^i allow us to determine that C has acquired the value 1 and R_1 has successively been set during this interval (transitions e_0^i and e_1^i, respectively). The contents of the registers at measurement m_3^i let us infer that R_1 has been reset (transition e_2^i) and that the value of R_2 has changed to 1 (transition e_3^i). We know that both e_2^i and e_3^i have taken place after e_1^i, but we have no information about the relative order of these transitions. Finally, m_4^i and m_5^i acknowledge that R_2 has successively been reset to 0 (e_4^i), and the same has then happened to C (e_5^i).

We will now give a formalization of this example and use various modal event calculi to draw conclusions about it. The situation relative to session i depicted in Figure 1 can be represented by the EC-structure $\mathcal{H}^i = (E^i,\ P,\ [\cdot\rangle^i,\ \langle\cdot]^i,\ ]\cdot,\cdot[)$, whose components are defined as follows:

- $E^i = \{e_0^i, e_1^i, e_2^i, e_3^i, e_4^i, e_5^i\}$;

- $P = \{C, R_1, R_2\}$;
- $[C\rangle^i = \{e_0^i\}$, $[R_1\rangle^i = \{e_1^i\}$, $[R_2\rangle^i = \{e_3^i\}$;
- $\langle C]^i = \{e_5^i\}$, $\langle R_1]^i = \{e_2^i\}$, $\langle R_2]^i = \{e_4^i\}$;
- $]\cdot,\cdot[= \emptyset$.

We have encoded transitions as events with the same name and denoted with R_j $(j = 1, 2)$ the property that register R_j has value 1, and similarly for C.

The ordering w of the transitions inferred from the measurements corresponds to the transitive closure of the following graph.

$$\begin{tikzcd}[row sep=small] & & e_2^i \arrow[dr] & & \\ e_0^i \arrow[r] & e_1^i \arrow[ur] \arrow[dr] & & e_4^i \arrow[r] & e_5^i \\ & & e_3^i \arrow[ur] & & \end{tikzcd}$$

Consider the formulas $\varphi^i = R_1(e_1^i, e_2^i) \ \wedge \ (e_2^i < e_3^i) \ \wedge \ R_2(e_3^i, e_4^i)$. In order to verify that the switch S is not faulty in session i, we must ensure that the registers R_1 and R_2 display the expected behavior in all refinements of the current knowledge state w. This amounts to proving that the *CMEC*-formula $\Box\varphi^i$ is true in w. If this is the case, there is no fault in session i, although other sessions might indicate that S is dysfunctional. If we want to determine the existence of at least one extension of w where the registers behave as displayed in Figure 1, we must verify the truth of $\Diamond\varphi^i$ in w. If this *CMEC*-formula is true, we cannot be sure whether S is faulty or not.

Since $\mathcal{I}_{\mathcal{H}^i}; w \models \Diamond\varphi^i$ and $\mathcal{I}_{\mathcal{H}^i}; w \not\models \Box\varphi^i$, a faulty behavior of S in session i is possible but not certain. Assume now that, unlike the actual situation depicted in Figure 1, we extend w so that e_3^i precedes e_2^i, call w_1 the resulting state. Then, $\mathcal{I}_{\mathcal{H}^i}; w_1 \not\models \Diamond\varphi^i$, that is, the evolution of the values in the registers hints at a fault. Conversely, let us refine w so that e_2^i precedes e_3^i, and call w_2 the resulting knowledge state. Then, we can infer $\mathcal{I}_{\mathcal{H}^i}; w_2 \models \Box\varphi^i$, and hence we can conclude that the switch S is certainly not faulty.

The formulas we have used so far belong to the language of *CMEC*. This is unfortunate since model checking is intractable in it. However, there exist equivalent formulas in the language of *ECMEC*, that can be checked in polynomial time. By the distributivity $\Box$ over $\wedge$, $\Box\varphi^i$ is equivalent to the *ECMEC*-formula:

$$\psi^i = \Box R_1(e_1^i, e_2^i) \wedge \Box(e_2^i < e_3^i) \wedge \Box R_2(e_3^i, e_4^i)$$

Therefore, we can use *ECMEC* and ψ^i to establish if the switch S is fault-free or is possibly defective.

Consider the following approximation of $\Diamond\varphi^i$:

$$\chi^i = \Diamond R_1(e_1^i, e_2^i) \wedge \Diamond(e_2^i < e_3^i) \wedge \Diamond R_2(e_3^i, e_4^i)$$

which, in general, is not equivalent to, but only implied by, $\Diamond\varphi^i$. It easy to see that the structure $\mathcal{H}$ satisfies the singleton condition (Section 2). Moreover, (e_1^i, e_2^i) and (e_3^i, e_4^i) belong to w, and thus, from Proposition 2.4, it follows that $\mathcal{I}_{\mathcal{H}^i}; w \models \chi^i$ if and only if $\mathcal{I}_{\mathcal{H}^i}; w \models \Diamond\varphi^i$. Therefore, we can use *ECMEC* and χ^i to establish whether the switch S is certainly faulty or is possibly correct.

Quantifiers allow extending this line of reasoning from session i to all recorded sessions, enabling us to give a faithful representation of the given rule for detecting faults in register S. We achieve this by using quantifiers to abstract from the specific events appearing in φ^i above. Consider the following *QCMEC*-formulas φ_1 and φ_2:

$$\begin{aligned}\alpha(\vec{x}) &= x_0 < x_1 \ \wedge \ R_1(x_1, x_2) \ \wedge \\ &\quad\; x_2 < x_3 \ \wedge \ R_2(x_3, x_4) \ \wedge \ x_4 < x_5 \\ \varphi_1 &= \forall x_0.\, \forall x_5.\, C(x_0, x_5) \supset \exists x_1.\, \exists x_2.\, \exists x_3.\, \exists x_4.\, \Box\alpha(\vec{x}) \\ \varphi_2 &= \forall x_0.\, \forall x_5.\, C(x_0, x_5) \supset \exists x_1.\, \exists x_2.\, \exists x_3.\, \exists x_4.\, \Diamond\alpha(\vec{x})\end{aligned}$$

where $\vec{x}$ stands for the list of variables $(x_0, x_1, x_2, x_3, x_4, x_5)$. Given a global EC-structure $\mathcal{H}$, we can be certain that the switch S is not faulty no matter how the order of actual transitions differs from what was inferred from the measurements if $\mathcal{I}_{\mathcal{H}}; w \models \varphi_1$ holds. On the other hand, the possibility that S behaves correctly is left open if $\mathcal{I}_{\mathcal{H}}; w \models \varphi_2$ is valid. Both φ_1 and φ_2 are in $\mathcal{L}_{\mathcal{H}}(QCMEC)$. However, if we distribute the modal operator over the boolean connectives in φ_1 and φ_2, we obtain two formulas, say φ_1' and φ_2', that are in the language of *EQCMEC* and thus can be solved in a time that is polynomial in the number of events (Section 5). It is possible to show that, for any $w' \in \mathrm{Ext}(w)$, $\mathcal{I}_{\mathcal{H}}; w' \models \varphi_1$ if and only if $\mathcal{I}_{\mathcal{H}}; w' \models \varphi_1'$ and $\mathcal{I}_{\mathcal{H}}; w' \models \varphi_2$ if and only if $\mathcal{I}_{\mathcal{H}}; w' \models \varphi_2'$.

4 Implementation

The Event Calculus [9] has traditionally been implemented in the logic programming language *Prolog* [13]. Recent extensions to *EC* have instead adopted λ*Prolog* [10] in order to achieve a declarative yet simple encoding, necessary to formally establish correctness issues [6]. In this section, we will rely again on λ*Prolog* to obtain an elegant encoding of *QCMEC* and to prove its correctness. Space reasons forbid discussing the implementation of its subcalculi.

4.1 λProlog in a nutshell

Due to space limitations, we shall assume the reader to be familiar with the logic programming language *Prolog* [13]. We will instead illustrate some of the characteristic constructs of λ*Prolog* at an intuitive level. We invite the interested reader to consult [10] for a more complete discussion, and [6] for a 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:

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

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)`").

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`.`". Syntax is provided for declaring infix symbols. Application and λ-abstraction can be typed if their subexpression satisfy certain constraints. 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 λ*Prolog* differs from *Prolog* for the availability of implication 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. The goal $\exists x. G$ is entered as "`sigma` $x \setminus G$". We will also take advantage of *negation-as-failure*, denoted `not`. We will not rely directly on the other powerful constructs offered by this language. 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 λ*Prolog* 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 [6, 10] for details.

λ*Prolog* 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.

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

4.2 Implementation of QCMEC in λProlog

We will now give an implementation of *QCMEC* in λ*Prolog*. The resulting module, called `qcmec`, is displayed in Appendix A. The rule to diagnose hardware faults and an example from Section 3 are included in Appendices B and C. This code has been tested using the *Terzo* implementation of λ*Prolog*, version 1.0b, which is available from

```
http://www.cse.psu.edu/~dale/lProlog/
```

We define a family of representation functions $\ulcorner \cdot \urcorner$ that relate the mathematical entities we have been using in Section 2 to terms in λ*Prolog*. Specifically, we will need to encode EC-structures, the associated orderings, and the language of $QCMEC$. In the remainder of this section, we will refer to a generic EC-structure $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$.

We represent $\mathcal{H}$ by giving an encoding of the entities that constitute it. We introduce the types `event` and `property` so that every event in E (property in P) is represented by a distinct constant of type `event` (of type `property`). Event variables are represented as λ*Prolog* 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.

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 encapsule the clauses for the predicate `before`, which implements its transitive closure, in the module `transClo`. We do not show details for space reasons, but a quadratic implementation can be found in [2].

In order to encode the syntax of *QCMEC*, we define the type `mvi`, intended to represent the formulas of this language (as opposed to the formulas of λ*Prolog*, that have type `o`). The representation of formulas is then relatively standard [6], except for quantifiers:

$$
\begin{aligned}
\ulcorner \bar{p}(\bar{e}_1, \bar{e}_2) \urcorner &= \texttt{period}\, \ulcorner \bar{e}_1 \urcorner\, \ulcorner \bar{p} \urcorner\, \ulcorner \bar{e}_2 \urcorner \\
\ulcorner \bar{e}_1 < \bar{e}_2 \urcorner &= \ulcorner \bar{e}_1 \urcorner\, \texttt{precedes}\, \ulcorner \bar{e}_2 \urcorner \\
\ulcorner \neg\varphi \urcorner &= \texttt{neg}\, \ulcorner \varphi \urcorner \\
\ulcorner \varphi_1 \wedge \varphi_2 \urcorner &= \ulcorner \varphi_1 \urcorner\, \texttt{and}\, \ulcorner \varphi_2 \urcorner \\
\ulcorner \varphi_1 \vee \varphi_2 \urcorner &= \ulcorner \varphi_1 \urcorner\, \texttt{or}\, \ulcorner \varphi_2 \urcorner \\
\ulcorner \varphi_1 \supset \varphi_2 \urcorner &= \ulcorner \varphi_1 \urcorner\, \texttt{implies}\, \ulcorner \varphi_2 \urcorner \\
\ulcorner \Box\varphi \urcorner &= \texttt{must}\, \ulcorner \varphi \urcorner \\
\ulcorner \Diamond\varphi \urcorner &= \texttt{may}\, \ulcorner \varphi \urcorner \\
\ulcorner \forall x.\, \varphi \urcorner &= \texttt{forAllEvent}\ (x \setminus \ulcorner \varphi \urcorner) \\
\ulcorner \exists x.\, \varphi \urcorner &= \texttt{forSomeEvent}\ (x \setminus \ulcorner \varphi \urcorner)
\end{aligned}
$$

Quantifiers differ from the other syntactic entities of a language such as *QCMEC* 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. λ*Prolog* and other higher-order languages permit a much leaner emulation since λ-abstraction $(x \setminus M)$ is itself a binder and their implementations come equipped with (efficient) ways of handling it. The idea, known as *higher-order abstract syntax* [10], is then to use λ*Prolog*'s abstraction mechanism as a universal binder. Binding constructs in the object language are then expressed as constants that take a λ-abstracted term as their argument (for example `forSomeEvent` is declared of type `(event -> mvi) -> mvi`). Variable renaming happens behind the scene, and substitution is delegated to the meta-language as β-reduction.

An example will shed some light on this technique. Consider the formula $\varphi = \exists x.\, p(x, e_2)$, which 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 clauses *12–13* in our implementation.

We represent the truth of a formula in *QCMEC* by means of the predicate `holds`. Clauses *1* to *13* in Appendix A implement the specification of this language given in Section 2. More precisely, clauses *1* and *2* provide a direct encoding of Definition 2.1, where clause *2* faithfully emulates the meta-predicate *br*. Clause *3* captures the meaning of the precedence construct, while clauses *4* to *7* reduce the truth check for the connectives of *QCMEC* to the derivability of the corresponding λ*Prolog* constructs. Notice that implication is translated back to a combination of negation and disjunction in clause *7*. Clauses *8* to *11* implement the semantics of the modalities as the recursive visit of all the extensions of the current knowledge state; further details can be found in [6]. Existential quantifiers are handled similarly to connectives in clause *12*. 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 *13*. A lengthy discussion of the logical reasons behind this step can be found in [6].

4.3 Soundness and Completeness

The encoding we have chosen as an implementation of *QCMEC* 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 definition of *QCMEC* given in Section 2, and the exploitation of the declarative features of λ*Prolog*.

We only show the statement of our soundness and completeness result since a fully worked out proof would require a very detailed account of the semantics of λ*Prolog*, and is rather long, although simple. Space constraints prevent us from doing so. The interested reader can find the full development of a proof that relies on the same techniques in [6].

Theorem 4.1 (*Soundness and completeness of* `qcmec`)

Let $\mathcal{H} = (E,\ P,\ [\cdot\rangle,\ \langle\cdot],\ ]\cdot,\cdot[)$ *be an EC-structure, o a binary acyclic relation over E and* φ *and formula in* $\mathcal{L}_{\mathcal{H}}(QCMEC)$*, then*

$$\texttt{qcmec}, \ulcorner\mathcal{H}\urcorner, \ulcorner o\urcorner \vdash \texttt{holds}\ \varphi \quad \textit{iff} \quad \mathcal{I}_{\mathcal{H}}; o^{+} \models \varphi. \qquad \blacksquare$$

5 Complexity Analysis

This section is dedicated to studying the complexity of the various modal event calculi presented in Section 2. We assume the reader familiar with computational complexity theory [12]. Given an *EC*-structure $\mathcal{H}$, a knowledge state $w \in W_{\mathcal{H}}$ and a formula φ relative to any of the modal event calculi presented in Section 2, we want to characterize the complexity of the problem of establishing whether $\mathcal{I}_{\mathcal{H}}; w \models \varphi$ is true, which is an instance of the general problem of model checking.

We model our analysis around the truth relations given in Definitions 2.2, 2.3 and 2.5. We measure the complexity of testing whether $\mathcal{I}_{\mathcal{H}}; w \models \varphi$ holds in terms of the size n of the input structure (where n is the number of recorded events) and the size k of the input formula (without loss of generality, we sometimes interpret k as the number of atomic formulas occurring in φ).

The notion of cost we adopt is as follows: we assume that verifying the truth of the propositions $e \in [p\rangle$ and $e \in \langle p]$ costs $\mathcal{O}(1)$. Although possible in principle, it is disadvantageous to maintain knowledge states as transitive relations. We instead record an acyclic binary relation o on events whose transitive closure o^+ is w. Verifying whether $e_1 <_w e_2$ holds becomes a reachability problem in o and it can be solved in quadratic time $O(n^2)$ in the number n of events [2]. The cost of solving the query $e_1 < e_2$ is therefore quadratic.

We begin our analysis from the plain Event Calculus. Model checking in *EC* (Definition 2.2) is a polynomial task and costs $\mathcal{O}(n^3)$ [2, 4].

Theorem 5.1 (*Cost of model checking in EC*)
Model checking in EC has complexity $\mathcal{O}(n^3)$. ■

We obtain the same bound if we allow property-labeled intervals $p(e_1, e_2)$, possibly prefixed with at most one modal operator [4]. This bound does not change if we consider precedence queries.

An *ECMEC*-formula is the boolean combination of a number of atomic formulas, i.e. property-labeled intervals $p(e_1, e_2)$ or precedence tests $e_1 < e_2$, possibly prefixed with a modal operator.

Given an *ECMEC*-formula that contains k atomic formulas, checking it reduces to testing k atomic formulas, possibly prefixed with a modal operator, each of them is solved in $\mathcal{O}(n^3)$. Thus, model checking in *ECMEC* has polynomial complexity.

Theorem 5.2 (*Cost of model checking in ECMEC*)
Model checking in ECMEC has complexity $\mathcal{O}(kn^3)$. ■

Model checking in *CMEC* (Definition 2.3) involves an exhaustive exploration of the extensions of the current knowledge state, whose number is, in general, exponential in the number of events. This raises complexity beyond tractability.

Theorem 5.3 (*Cost of model checking in CMEC*)
Model checking in CMEC is **PSPACE**-*complete.*

Proof. In order to prove that model checking in *CMEC* 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 *CMEC*-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 $\varphi = p(e_1, e_2)$ (resp. $\varphi = \neg p(e_1, e_2)$), the algorithm accepts it if and only if all points (resp. at least one point) from *i* to *iv* of Definition 2.2 hold (resp. does not hold). Finally, if $\varphi = e_1 < e_2$ (resp. $\varphi = \neg(e_1 < e_2)$), the algorithm accepts if and only if $e_1 <_w e_2$ (resp. $e_1 \not<_w e_2$).

It follows, from the definition of acceptance of alternating machines [12], that a *CMEC*-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 *CMEC* in **AP**. Since **AP** = **PSPACE** [12], it is in **PSPACE**.

In order to prove that the considered problem is **PSPACE**-hard, we define a (polynomial) reduction of QSAT [12] into *CMEC*.

Let $G = \exists x_1. \forall x_2. \exists x_3. \forall x_4. \dots Qx_n. F(x_1, x_2, \dots x_n)$, with $n \geq 1$, be a quantified Boolean formula where the quantifiers alternate, so that Q is $\exists$ ($\forall$) if n is odd (even).

We now define the EC-structure $\mathcal{H} = (E,\, P,\, [\cdot\rangle,\, \langle\cdot],\,]\cdot,\cdot[)$ such that:

- $E = \{e_{x_i}, e_{\neg x_i} : 1 \leq i \leq n\}$;
- $P = \{p_{x_i} : 1 \leq i \leq n\}$;
- $[p_{x_i}\rangle = \{e_{x_i}\}$, $\langle p_{x_i}] = \{e_{\neg x_i}\}$, for $1 \leq i \leq n$;
- $]\cdot,\cdot[= \emptyset$.

Moreover, let $w = \emptyset$, and $\hat{F}$ be the formula obtained replacing in the formula $F(x_1, x_2, \dots x_n)$ every occurrence of a variable x_i with $p_{x_i}(e_{x_i}, e_{\neg x_i})$.

Let $V = \{x_1, x_2, ...x_n\}$ be the set of variables of G. A knowledge state w in $W_{\mathcal{H}}$ naturally induces an assignment t_w for the variables in V such that, for every x in V, $t_w(x)$ is true if and only if $w \models p_x(e_x, e_{\neg x})$. Since only the relative order of the events e_x and $e_{\neg x}$, for every variable x in V, is relevant, different knowledge states may induce the same assignment. This many-to-one correspondence between knowledge states and assignments establishes an equivalence relation on the set of all knowledge states for $\mathcal{H}$ such that the truth of a propositional CMEC-formula on $\mathcal{H}$ (i.e. , a CMEC-formula on $\mathcal{H}$ without modalities) is invariant within each equivalence class.

Further, consider the following recursive definition of the $CMEC$-formula F_k:

$$F_k = \begin{cases} \Diamond \hat{F} & k = n \\ \Diamond(\psi_{k+1} \wedge \Box \hat{F}) & k = n-1 \\ \Diamond(\psi_{k+1} \wedge \Box(\psi_{k+2} \supset F_{k+2})) & \text{otherwise} \end{cases}$$

where

$$\psi_k = \textstyle\bigwedge_{k \leq i \leq n} \neg(e_{x_i} < e_{\neg x_i}) \wedge \neg(e_{\neg x_i} < e_{x_i}).$$

Observe that, if $w \models \psi_k$, then, for every i from k to n, the events e_{x_i} and $e_{\neg x_i}$ are unordered.

It is possible to prove that, for $\varphi = F_1$, $w \models \varphi$ if and only if G is true. The basic step of the proof is the mapping of the propositional quantifiers of G into the modal operators of F_1. However, the scope of a quantifier is limited to the variable that it quantifies, whereas the scope of a modal operator in the formula $\Box F$ or $\Diamond F$ extends to the whole formula F. To cope with this problem, we take advantage of the formulas ψ_k in order to restore the correct context for the evaluation of F. ■

In the following, we analyze the complexity of the quantified calculi defined in Section 2. We begin our analysis with the complexity of *EQCMEC*, i.e. the quantified version of *ECMEC*. We have proved that model checking in *ECMEC* is polynomial time-bounded (Theorem 5.2). However, the extension of *ECMEC* with quantifiers arises complexity beyond **P**. In particular, model checking in *EQCMEC* is **PSPACE**-complete, as proved by the following theorem.

Theorem 5.4 (*Cost of model checking in EQCMEC*)
Model checking in EQCMEC is **PSPACE***-complete.*

Proof. In order to prove that model checking in *EQCMEC* is in **PSPACE**, we show that it belongs to **AP**. In order to do so, we extend the alternating polynomial time algorithm used in the proof of Theorem 5.3. 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. $\forall x.\,\neg\alpha$).

From the definition of acceptance of alternating machines [12], it follows that an *EQCMEC*-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** [12], it is in **PSPACE**.

In order to prove that the considered problem is **PSPACE**-hard, we define a (polynomial) reduction of QSAT [12] into *EQCMEC*.

Let $G = \exists x_1.\,\forall x_2.\,\exists x_3.\,\forall x_4.\,\ldots Qx_n.\,F(x_1, x_2, \ldots x_n)$, with $n \geq 1$, be a quantified boolean formula where the quantifiers alternate (so that Q is $\exists$ ($\forall$) if n is odd (even)).

We then define an *EC*-structure $\mathcal{H} = (E,\, P,\, [\cdot\rangle,\, \langle\cdot],\,]\cdot,\cdot[)$ such that:

- $E = \{e_1, e_2, \ldots e_n, e\}$;
- $P = \{p_1, p_2, \ldots p_n\}$;
- $[p_i\rangle = \{e_i\}$ and $\langle p_i] = \{e\}$, for $1 \leq i \leq n$;
- $]\cdot,\cdot[= \emptyset$.

Moreover, let $w = \{(e_i, e) : 1 \leq i \leq n\}\}$, and

$$\varphi = \exists x_1.\,\forall x_2.\,\exists x_3.\,\forall x_4.\,\ldots Q_n x_n.\,\hat{F}(x_1, x_2, \ldots x_n)$$

where $\hat{F}(x_1, x_2, \ldots x_n)$ is obtained replacing every occurrence of a variable x_i in the formula $F(x_1, x_2, \ldots x_n)$ with $p_i(x_i, e)$. Notice that φ is an *EQCMEC*-formula. In particular, it has no modal operator. It is not difficult to see that $w \models \varphi$ if and only if G is true. ■

In the following, we characterize the explicit time complexity of model checking in *EQCMEC*. Since model checking in *EQCMEC* is **PSPACE**-complete, we expect an exponential bound.

We will exploit the unfolding lemma (2.6). This result affirms that every formula involving one event quantifier at its top-level can be replaced with the conjunction or disjunction of n instances of it, where n is the number of events. If we have a nesting of q such quantifiers, we are led to solve n^q instances. In general, if we eliminate in this manner all event quantifiers in a formula φ of size k, we will produce a formula φ' without quantifiers, i.e. an *ECMEC* formula, of size at most kn^k. Taking advantage of Theorem 5.2, we get the following upper bound.

Theorem 5.5 (*Upper bound for complexity of model checking in EQCMEC*)
Model checking in EQCMEC has complexity $\mathcal{O}(kn^{k+3})$. ■

Practical applications using modal event calculi with quantifiers are expected to model situations involving a large number of events, while the size of the queries will in general be limited. The hardware fault diagnosis example in Section 3 falls into this category. In such contexts, the fact that *EQCMEC* is polynomial in the number of events is essential. At worst, the dependence of the exponent on k may lead to polynomials of high degree.

Finally, we consider the calculus *QCMEC*. Since *EQCMEC* is a linguistic fragment of *QCMEC*, model checking in *QCMEC* is **PSPACE**-hard. Nevertheless, it is possible to show that model checking in *QCMEC* is polynomial space-bounded.

Theorem 5.6 (*Cost of model checking in QCMEC*)
Model checking in QCMEC is **PSPACE***-complete.*

Proof. To see that the considered problem is in **PSPACE**, we exploit the polynomial alternating algorithm defined in the proof of Theorem 5.4 for *EQCMEC*. It works correctly for *QCMEC* as well.

Since *EQCMEC* is a linguistic fragment of *QCMEC* and *EQCMEC* is **PSPACE**-hard (Theorem 5.4), model checking in *QCMEC* is **PSPACE**-hard. ■

In Section 4 we have transliterated the definition of *QCMEC* and its subcalculi in the higher-order logic programming language λ*Prolog* [10].

6 Conclusions and Future Work

In this paper, we have extended a number of modal event calculi [1, 4, 9] with the possibility of using quantifiers and precedence tests in queries. The net effect of these combined additions has been a substantial gain in expressiveness. The extra computational cost was shown acceptable for queries of a reasonable size in those subcalculi that are tractable without quantifiers. We have implemented the resulting formalisms in the higher-order logic programming language λ*Prolog* [10], which we used to encode case studies from the area of hardware and medical diagnosis.

We restricted our attention to complete actual courses of events, whose temporal ordering, specified by means of a set of ordered pairs of events, is only partially known. The considered scenarios are thus characterized by a set of occurred events, which is fixed once and for all and does not include concurrent or divisible events, and by incomplete (but consistent) information about an actual total order, given in terms of the relative order of pairs of event occurrences. However, most of these restrictions can actually be relaxed:

- completeness of the set of event occurrences (we can have incomplete knowledge about both the event occurrences and their temporal ordering);
- occurred events only (the events can be either actual events or hypothetical ones).

All the proved results, indeed, do not rely on such restrictions.

Finally, we intend gaining a better understanding of the interactions among the various operators of our calculi, in particular between quantifiers and modalities, in order to devise simplifications of costly queries and thus better implementations. We also intend studying the integration of preconditions [5].

Acknowledgments

We would like to thank Paolo Liberatore, Peter Jonsson and Rob Miller for their useful comments.

The first author was supported by ONR grant N00014-97-1-0505, Multidisciplinary University Research Initiative *Semantic Consistency in Information Exchange*. The work of the third author was partially supported by the CNR project *Programmazione logica: strumenti per analisi e trasformazione di programmi; tecniche di ingegneria del software; estensioni con vincoli, concorrenza ed oggetti (STE).*

References

[1] 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.

[2] Iliano Cervesato, Luca Chittaro, and Angelo Montanari. 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.

[3] 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.

[4] 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. To appear.

[5] Iliano Cervesato, Massimo Franceschet, and Angelo Montanari. Modal event calculi with preconditions. In R. Morris and L. Khatib, editors, *Fourth International Workshop on Temporal Representation and Reasoning — TIME'97*, pages 38–45, Daytona Beach, FL, 10–11 May 1997. IEEE Computer Society Press.

[6] Iliano Cervesato and Angelo Montanari. A general modal framework for the event calculus and its skeptical and credulous variants. In *Journal of Logic Programming*, 1998. To appear.

[7] 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.

[8] 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.

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

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

[11] K. Nökel. *Temporarily Distributed Symptoms in Technical Diagnosis*. Springer-Verlag, 1991.

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

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

A Implementation of *QCMEC*

```
module qcmec.
accumulate transClo.

kind event    type.
kind property type.
kind mvi      type.

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

% ------- MVIs
type period  event -> property -> event -> mvi.
type holds   mvi -> o.
type broken  event -> property -> event -> o.

holds (period Ei P Et) :-                              % 1 %
        happens Ei, initiates Ei P,
        happens Et, terminates Et P,
        before Ei Et, not (broken Ei P Et).
broken Ei P Et :-                                      % 2 %
        happens E,
        before Ei E, before E Et,
        (initiates E Q; terminates E Q),
        (exclusive P Q; P = Q).

% ------- Ordering
type precedes  event -> event -> mvi.  infixr precedes 6.

holds (E1 precedes E2) :-  before E1 E2.               % 3 %

% ------- Connectives
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).                       % 4 %
holds (X and Y) :-  holds X, holds Y.                  % 5 %
holds (X or Y) :-  holds X; holds Y.                   % 6 %
holds (X implies Y) :-  holds ((neg X) or Y).          % 7 %

% ------- Modalities
type must        mvi -> mvi.
type may         mvi -> mvi.
type fails_must  mvi -> o.

holds (must X) :-  holds X, not (fails_must X).        % 8 %

fails_must X :-                                        % 9 %
        happens E1, happens E2, not (E1 = E2),
        not (before E1 E2), not (before E2 E1),
        beforefact E1 E2 => not (holds (must X)).

holds (may X) :-  holds X.                            % 10 %
holds (may X) :-                                      % 11 %
        happens E1, happens E2, not (E1 = E2),
        not (before E1 E2), not (before E2 E1),
        beforefact E1 E2 => holds (may X).

% ------- Quantifiers
```

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

holds (forAllEvent X) :-                                 % 12 %
        not (sigma E \ (happens E, not (holds (X E)))).
holds (forSomeEvent X) :-  sigma E \ holds (X E).        % 13 %
```

B Diagnosing Hardware Faults

```
module cncc.
accumulate qcmec.

type c  property.
type r1 property.
type r2 property.
type phi1 o.
type phi2 o.

phi1 :- holds (forAllEvent E0 \
               forAllEvent E5 \
                 ((period E0 c E5) implies
                    (forSomeEvent E1 \
                     forSomeEvent E2 \
                     forSomeEvent E3 \
                     forSomeEvent E4 \
                       (must ((E0 precedes E1)  and
                              (period E1 r1 E2) and
                              (E2 precedes E3)  and
                              (period E3 r2 E4) and
                              (E4 precedes E5)))))).

phi2 :- holds (forAllEvent E0 \
               forAllEvent E5 \
                 ((period E0 c E5) implies
                    (forSomeEvent E1 \
                     forSomeEvent E2 \
                     forSomeEvent E3 \
                     forSomeEvent E4 \
                       (may ((E0 precedes E1)  and
                             (period E1 r1 E2) and
                             (E2 precedes E3)  and
                             (period E3 r2 E4) and
                             (E4 precedes E5)))))).
```

C A Specific Situation

```
module example.
accumulate cncc.

type e0 event.    happens e0.    initiates  e0 c.
type e1 event.    happens e1.    initiates  e1 r1.
type e2 event.    happens e2.    terminates e2 r1.
type e3 event.    happens e3.    initiates  e3 r2.
type e4 event.    happens e4.    terminates e4 r2.
type e5 event.    happens e5.    terminates e5 c.

beforefact e0 e1.    beforefact e1 e2.    beforefact e1 e3.
beforefact e2 e4.    beforefact e3 e4.    beforefact e4 e5.
```